Contents

- Windows kernel
  - Aux_klib.h
    - AUX_MODULE_BASIC_INFO structure
    - AUX_MODULE_EXTENDED_INFO structure
    - KBUGCHECK_DATA structure
    - AuxKlibEnumerateSystemFirmwareTables function
    - AuxKlibGetBugCheckData function
    - AuxKlibGetImageExportDirectory function
    - AuxKlibGetSystemFirmwareTable function
    - AuxKlibInitialize function
    - AuxKlibQueryModuleInformation function
  - Buffring.h
  - Ioaccess.h
    - READ_PORT_UCHAR function
    - READ_PORT_ULONG function
    - READ_PORT_USHORT function
    - READ_REGISTER_UCHAR function
    - READ_REGISTER_ULONG function
    - READ_REGISTER_USHORT function
    - WRITE_PORT_UCHAR function
    - WRITE_PORT_ULONG function
    - WRITE_PORT_USHORT function
    - WRITE_REGISTER_UCHAR function
    - WRITE_REGISTER_ULONG function
    - WRITE_REGISTER_USHORT function
  - Iointex.h
    - WdmlibIoConnectInterruptEx function
    - WdmlibIoDisconnectInterruptEx function
    - WdmlibIoGetAffinityInterrupt function

Miniport.h

- BUS_DATA_TYPE enumeration
- CM_EISA_FUNCTION_INFORMATION structure
- CM_EISA_SLOT_INFORMATION structure
- CM_MCA_POS_DATA structure
- GROUP_AFFINITY structure
- INTERFACE structure
- INTERFACE_TYPE enumeration
- IO_RESOURCE_DESCRIPTOR structure
- IO_RESOURCE_LIST structure
- IO_RESOURCE_REQUIREMENTS_LIST structure
- IRQ_DEVICE_POLICY enumeration
- IRQ_PRIORITY enumeration
- KINTERRUPT_MODE enumeration
- KINTERRUPT_POLARITY enumeration
- MEMORY_CACHING_TYPE enumeration
- PCI_COMMON_CONFIG structure
- PCI_SLOT_NUMBER structure
- PROCESSOR_NUMBER structure
- FIELD_OFFSET macro
- InterlockedAnd function
- InterlockedCompareExchange function
- InterlockedCompareExchangePointer function
- InterlockedDecrement function
- InterlockedExchange function
- InterlockedExchangeAdd function
- InterlockedExchangePointer function
- InterlockedIncrement function
- InterlockedOr function
- InterlockedXor function
- PINTERFACE_DEREFERENCE callback function
- PINTERFACE_REFERENCE callback function

READ_PORT_BUFFER_UCHAR function
READ_PORT_BUFFER_ULONG function
READ_PORT_BUFFER_USHORT function
READ_PORT_UCHAR function
READ_PORT_ULONG function
READ_PORT_USHORT function
READ_REGISTER_BUFFER_UCHAR function
READ_REGISTER_BUFFER_ULONG function
READ_REGISTER_BUFFER_USHORT function
READ_REGISTER_UCHAR function
READ_REGISTER_ULONG function
READ_REGISTER_USHORT function
WRITE_PORT_BUFFER_UCHAR function
WRITE_PORT_BUFFER_ULONG function
WRITE_PORT_BUFFER_USHORT function
WRITE_PORT_UCHAR function
WRITE_PORT_ULONG function
WRITE_PORT_USHORT function
WRITE_REGISTER_BUFFER_UCHAR function
WRITE_REGISTER_BUFFER_ULONG function
WRITE_REGISTER_BUFFER_USHORT function
WRITE_REGISTER_UCHAR function
WRITE_REGISTER_ULONG function
WRITE_REGISTER_USHORT function

Ntddk.h

BDCB_CALLBACK_TYPE enumeration
BDCB_CLASSIFICATION enumeration
BDCB_IMAGE_INFORMATION structure
BDCB_STATUS_UPDATE_CONTEXT structure
BDCB_STATUS_UPDATE_TYPE enumeration
BUS_DATA_TYPE enumeration
CONTROLLER_OBJECT structure

FILE_ALIGNMENT_INFORMATION structure

FILE_ATTRIBUTE_TAG_INFORMATION structure

FILE_DISPOSITION_INFORMATION structure

FILE_END_OF_FILE_INFORMATION structure

FILE_NAME_INFORMATION structure

FILE_VALID_DATA_LENGTH_INFORMATION structure

HARDWARE_COUNTER structure

HARDWARE_COUNTER_TYPE enumeration

IMAGE_INFO structure

IMAGE_INFO_EX structure

KEY_CACHED_INFORMATION structure

KEY_NAME_INFORMATION structure

KEY_VIRTUALIZATION_INFORMATION structure

MM_COPY_ADDRESS structure

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE enumeration

PHYSICAL_COUNTER_RESOURCE_LIST structure

PNP_LOCATION_INTERFACE structure

POWER_THROTTLING_PROCESS_STATE structure

POWER_THROTTLING_THREAD_STATE structure

PROCESS_MITIGATION_CHILD_PROCESS_POLICY structure

PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY structure

PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY structure

PS_CREATE_NOTIFY_INFO structure

PSCREATEPROCESSNOTIFYTYPE enumeration

PSCREATETHREADNOTIFYTYPE enumeration

SILO_MONITOR_REGISTRATION structure

SUBSYSTEM_INFORMATION_TYPE enumeration

CORRELATION_VECTOR structure

DRIVER_REINITIALIZE callback function

ExFreePool macro

EXPAND_STACK_CALLOUT callback function

ExRaiseAccessViolation function

ExRaiseDatatypeMisalignment function

ExUuidCreate function

HalAllocateHardwareCounters function

HalExamineMBR function

HalFreeHardwareCounters function

IoAllocateController function

IoAssignArcName macro

IoClearActivityIdThread function

IoCreateController function

IoDeassignArcName macro

IoDecrementKeepAliveCount function

IoDeleteController function

IoFreeController function

IoGetActivityIdIrp function

IoGetActivityIdThread function

IoGetConfigurationInformation function

IoGetFileObjectGenericMapping function

IoGetInitiatorProcess function

IoGetPagingIoPriority function

IoIncrementKeepAliveCount function

IoIsValidIrpStatus function

IoMakeAssociatedIrp function

IoPropagateActivityIdToThread function

IoQueryFullDriverPath function

IoRaiseHardError function

IoRaiseInformationalHardError function

IoRegisterBootDriverCallback function

IoRegisterBootDriverReinitialization function

IoRegisterDriverReinitialization function

IoReportDetectedDevice function

IoReportResourceForDetection function

IoReportRootDevice function

IoSetActivityIdIrp function

IoSetActivityIdThread function

IoSetHardErrorOrVerifyDevice function

IoSetMasterIrpStatus function

IoSetSystemPartition function

IoSetThreadHardErrorMode function

IoTransferActivityId function

IoUnregisterBootDriverCallback function

IoVerifyPartitionTable function

IoVolumeDeviceToDosName function

KeAreApcsDisabled function

KeBugCheck function

KeEnterCriticalRegion function

KeEnterGuardedRegion function

KeExpandKernelStackAndCallout function

KeGetCurrentNodeNumber function

KeGetCurrentProcessorNumber function

KeGetCurrentProcessorNumberEx function

KeInitializeCrashDumpHeader function

KeInvalidateAllCaches function

KeLeaveCriticalRegion function

KeLeaveGuardedRegion function

KePulseEvent function

KeQueryActiveGroupCount function

KeQueryActiveProcessorCount function

KeQueryActiveProcessorCountEx function

KeQueryActiveProcessors function

KeQueryGroupAffinity function

KeQueryHardwareCounterConfiguration function

KeQueryHighestNodeNumber function

KeQueryLogicalProcessorRelationship function

KeQueryMaximumGroupCount function

KeQueryMaximumProcessorCount function

KeQueryMaximumProcessorCountEx function

KeQueryNodeActiveAffinity function

KeQueryNodeMaximumProcessorCount function

KeQueryTickCount macro

KeRaiseIrqlToDpcLevel function

KeSetBasePriorityThread function

KeSetHardwareCounterConfiguration function

KeSetImportanceDpc function

KeSetTargetProcessorDpc function

MmAllocateContiguousMemory function

MmAllocateContiguousMemorySpecifyCache function

MmAllocateContiguousMemorySpecifyCacheNode function

MmAllocateContiguousNodeMemory function

MmAllocateNonCachedMemory function

MmCopyMemory function

MmFreeContiguousMemory function

MmFreeContiguousMemorySpecifyCache function

MmFreeNonCachedMemory function

MmGetPhysicalAddress function

MmIsAddressValid function

MmIsThisAnNtAsSystem function

MmLockPagableSectionByHandle function

MmSecureVirtualMemory function

MmUnsecureVirtualMemory function

NtOpenProcess function

PCREATE_PROCESS_NOTIFY_ROUTINE callback function

PCREATE_PROCESS_NOTIFY_ROUTINE_EX callback function

PCREATE_THREAD_NOTIFY_ROUTINE callback function

PGET_LOCATION_STRING callback function

pHalExamineMBR callback function

PLOAD_IMAGE_NOTIFY_ROUTINE callback function

PsAllocSiloContextSlot function

PsAttachSiloToCurrentThread function

PsCreateSiloContext function

PsDereferenceSiloContext function

PsDetachSiloFromCurrentThread function

PsFreeSiloContextSlot function

PsGetCurrentProcessId function

PsGetCurrentServerSilo function

PsGetCurrentSilo function

PsGetCurrentThread function

PsGetCurrentThreadId function

PsGetCurrentThreadTeb function

PsGetEffectiveServerSilo function

PsGetHostSilo function

PsGetJobServerSilo function

PsGetJobSilo function

PsGetParentSilo function

PsGetPermanentSiloContext function

PsGetProcessCreateTimeQuadPart function

PsGetProcessId function

PsGetServerSiloActiveConsoleId function

PsGetSiloContext function

PsGetSiloMonitorContextSlot function

PsInsertPermanentSiloContext function

PsInsertSiloContext function

PsIsHostSilo function

PsMakeSiloContextPermanent function

PsReferenceSiloContext function

PsRegisterSiloMonitor function

PsRemoveCreateThreadNotifyRoutine function

PsRemoveLoadImageNotifyRoutine function

PsRemoveSiloContext function

PsReplaceSiloContext function

PsSetCreateProcessNotifyRoutine function

PsSetCreateProcessNotifyRoutineEx function

PsSetCreateProcessNotifyRoutineEx2 function

PsSetCreateThreadNotifyRoutine function

PsSetCreateThreadNotifyRoutineEx function

PsSetLoadImageNotifyRoutine function

PsSetLoadImageNotifyRoutineEx function

PsStartSiloMonitor function

PsTerminateServerSilo function

PsUnregisterSiloMonitor function

RTL_RUN_ONCE_INIT_FN callback function

RtlCharToInteger function

RtlCompareString function

RtlConvertLongToLuid function

RtlConvertUlongToLuid function

RtlCopyString function

RtlEqualString function

RtlExtendCorrelationVector function

RtlGetEnabledExtendedFeatures function

RtlIncrementCorrelationVector function

RtlInitializeCorrelationVector function

RtlIsStateSeparationEnabled function

RtlMapGenericMask function

RtlPrefixUnicodeString function

RtlQueryRegistryValueWithFallback function

RtlRunOnceBeginInitialize function

RtlRunOnceComplete function

RtlRunOnceExecuteOnce function

RtlRunOnceInitialize function

RtlUpcaseUnicodeString function

- RtlUpperChar function
- RtlUpperString function
- RtlValidateCorrelationVector function
- RtlVolumeDeviceToDosName function
- SeSinglePrivilegeCheck function
- SILO_CONTEXT_CLEANUP_CALLBACK callback function
- SILO_MONITOR_CREATE_CALLBACK callback function
- SILO_MONITOR_TERMINATE_CALLBACK callback function
- ZwAllocateLocallyUniqueId function
- ZwDeviceIoControlFile function
- ZwOpenProcess function
- ZwPowerInformation function
- ZwQueryVolumeInformationFile function
- ZwSetInformationThread function
- ZwTerminateProcess function

Ntddsfio.h

- IO_ACCESS_MODE enumeration
- IO_ACCESS_TYPE enumeration

Ntddsysenv.h

- SYSENV_VALUE structure
- SYSENV_VARIABLE structure
- SYSENV_VARIABLE_INFO structure
- XVARIABLE_NAME structure
- XVARIABLE_NAME_AND_VALUE structure
- IOCTL_SYSENV_ENUM_VARIABLES IOCTL
- IOCTL_SYSENV_GET_VARIABLE IOCTL
- IOCTL_SYSENV_QUERY_VARIABLE_INFO IOCTL
- IOCTL_SYSENV_SET_VARIABLE IOCTL

Ntifs.h

- FILE_ACCESS_INFORMATION structure
- FILE_ALL_INFORMATION structure
- FILE_EA_INFORMATION structure

FILE_MODE_INFORMATION structure

MEMORY_BASIC_INFORMATION structure

MEMORY_INFORMATION_CLASS enumeration

IoGetAttachedDeviceReference function

IoGetConfigurationInformation function

IoSetStartIoAttributes function

IoSizeOfIrp macro

IoStartNextPacket function

IoStartNextPacketByKey function

IoStartPacket function

IoStartTimer function

IoStopTimer function

IoWriteErrorLogEntry function

KeGetProcessorIndexFromNumber function

KeGetProcessorNumberFromIndex function

KeQueryPerformanceCounter function

KeSetKernelStackSwapEnable function

KeStallExecutionProcessor function

NtAllocateVirtualMemory function

NtClose function

NtCreateFile function

NtCreateSection function

NtDeviceIoControlFile function

NtDuplicateToken function

NtFlushBuffersFileEx function

NtFreeVirtualMemory function

NtFsControlFile function

NtLockFile function

NtOpenFile function

NtOpenProcessTokenEx function

NtOpenThreadTokenEx function

NtQueryDirectoryFile function

NtQueryInformationFile function

NtQueryInformationToken function

NtQueryObject function

NtQueryQuotaInformationFile function

NtQuerySecurityObject function

NtQueryVirtualMemory function

NtQueryVolumeInformationFile function

NtReadFile function

NtSetInformationFile function

NtSetInformationThread function

NtSetInformationToken function

NtSetQuotaInformationFile function

NtSetSecurityObject function

NtUnlockFile function

NtWriteFile function

PoCallDriver function

PoClearPowerRequest function

PoCreatePowerRequest function

PoDeletePowerRequest function

PoEndDeviceBusy function

PoQueryWatchdogTime function

PoRegisterDeviceForIdleDetection function

PoRegisterPowerSettingCallback function

PoRegisterSystemState function

PoSetDeviceBusyEx function

PoSetPowerRequest function

PoSetPowerState function

PoStartDeviceBusy function

PoStartNextPowerIrp function

PoUnregisterPowerSettingCallback function

PoUnregisterSystemState function

PsGetCurrentThread function

PsIsSystemThread function
RtlInitStringEx function
RtlUnicodeToUTF8N function
RtlUTF8ToUnicodeN function
SeFreePrivileges function
Ntimage.h
Ntintsafe.h
RtlByteToChar function
RtlByteToInt8 function
RtlDWordPtrAdd function
RtlDWordPtrMult function
RtlDWordPtrSub function
RtlInt8Add function
RtlInt8Mult function
RtlInt8Sub function
RtlInt8ToUChar function
RtlInt8ToUInt function
RtlInt8ToUInt8 function
RtlInt8ToUIntPtr function
RtlInt8ToULong function
RtlInt8ToULongLong function
RtlInt8ToULongPtr function
RtlInt8ToUShort function
RtlIntAdd function
RtlIntMult function
RtlIntPtrAdd function
RtlIntPtrMult function
RtlIntPtrSub function
RtlIntPtrToChar function
RtlIntPtrToInt function
RtlIntPtrToInt8 function
RtlIntPtrToLong function

RtlIntPtrToLongPtr function

RtlIntPtrToShort function

RtlIntPtrToUChar function

RtlIntPtrToUInt function

RtlIntPtrToUInt8 function

RtlIntPtrToUIntPtr function

RtlIntPtrToULong function

RtlIntPtrToULongLong function

RtlIntPtrToULongPtr function

RtlIntPtrToUShort function

RtlIntSub function

RtlIntToChar function

RtlIntToInt8 function

RtlIntToShort function

RtlIntToUChar function

RtlIntToUInt function

RtlIntToUInt8 function

RtlIntToULong function

RtlIntToULongLong function

RtlIntToUShort function

RtlLongAdd function

RtlLongLongAdd function

RtlLongLongMult function

RtlLongLongSub function

RtlLongLongToChar function

RtlLongLongToInt function

RtlLongLongToInt8 function

RtlLongLongToIntPtr function

RtlLongLongToLong function

RtlLongLongToLongPtr function

RtlLongLongToShort function

RtlLongLongToUChar function

RtlLongLongToUInt function

RtlLongLongToUInt8 function

RtlLongLongToULong function

RtlLongLongToULongLong function

RtlLongLongToUShort function

RtlLongMult function

RtlLongPtrAdd function

RtlLongPtrMult function

RtlLongPtrSub function

RtlLongPtrToChar function

RtlLongPtrToInt function

RtlLongPtrToInt8 function

RtlLongPtrToIntPtr function

RtlLongPtrToLong function

RtlLongPtrToShort function

RtlLongPtrToUChar function

RtlLongPtrToUInt function

RtlLongPtrToUInt8 function

RtlLongPtrToUIntPtr function

RtlLongPtrToULong function

RtlLongPtrToULongLong function

RtlLongPtrToULongPtr function

RtlLongPtrToUShort function

RtlLongSub function

RtlLongToChar function

RtlLongToInt function

RtlLongToInt8 function

RtlLongToIntPtr function

RtlLongToShort function

RtlLongToUChar function

RtlLongToUInt function

RtlLongToUInt8 function

RtlLongToUIntPtr function

RtlLongToULong function

RtlLongToULongLong function

RtlLongToULongPtr function

RtlLongToUShort function

RtlPtrdiffTAdd function

RtlPtrdiffTMult function

RtlPtrdiffTSub function

RtlShortAdd function

RtlShortMult function

RtlShortSub function

RtlShortToChar function

RtlShortToDWordPtr function

RtlShortToInt8 function

RtlShortToUChar function

RtlShortToUInt function

RtlShortToUInt8 function

RtlShortToUIntPtr function

RtlShortToULong function

RtlShortToULongLong function

RtlShortToULongPtr function

RtlShortToUShort function

RtlSizeTAdd function

RtlSizeTMult function

RtlSizeTSub function

RtlSSIZETAdd function

RtlSSIZETMult function

RtlSSIZETSub function

RtlUInt8Add function

RtlUInt8Mult function

RtlUInt8Sub function

RtlUInt8ToChar function

RtlUInt8ToInt8 function

RtlUIntAdd function

RtlUIntMult function

RtlUIntPtrAdd function

RtlUIntPtrMult function

RtlUIntPtrSub function

RtlUIntPtrToChar function

RtlUIntPtrToInt function

RtlUIntPtrToInt16 function

RtlUIntPtrToInt8 function

RtlUIntPtrToIntPtr function

RtlUIntPtrToLong function

RtlUIntPtrToLongLong function

RtlUIntPtrToLongPtr function

RtlUIntPtrToShort function

RtlUIntPtrToUChar function

RtlUIntPtrToUInt function

RtlUIntPtrToUInt16 function

RtlUIntPtrToUInt8 function

RtlUIntPtrToULong function

RtlUIntPtrToUShort function

RtlUIntSub function

RtlUIntToChar function

RtlUIntToInt function

RtlUIntToInt8 function

RtlUIntToIntPtr function

RtlUIntToLong function

RtlUIntToLongPtr function

RtlUIntToShort function

RtlUIntToUChar function

RtlUIntToUInt8 function

RtlUIntToUShort function

RtlULongAdd function

RtlULongLongAdd function

RtlULongLongMult function

RtlULongLongSub function

RtlULongLongToChar function

RtlULongLongToInt function

RtlULongLongToInt8 function

RtlULongLongToLong function

RtlULongLongToLongLong function

RtlULongLongToLongPtr function

RtlULongLongToShort function

RtlULongLongToUChar function

RtlULongLongToUInt function

RtlULongLongToUInt8 function

RtlULongLongToUIntPtr function

RtlULongLongToULong function

RtlULongLongToULongPtr function

RtlULongLongToUShort function

RtlULongMult function

RtlULongPtrAdd function

RtlULongPtrMult function

RtlULongPtrSub function

RtlULongPtrToChar function

RtlULongPtrToInt function

RtlULongPtrToInt8 function

RtlULongPtrToIntPtr function

RtlULongPtrToLong function

RtlULongPtrToLongLong function

RtlULongPtrToLongPtr function

RtlULongPtrToShort function

RtlULongPtrToUChar function

RtlULongPtrToUInt function

- RtlULongPtrToUInt8 function
- RtlULongPtrToUIntPtr function
- RtlULongPtrToULong function
- RtlULongPtrToUShort function
- RtlULongSub function
- RtlULongToChar function
- RtlULongToInt function
- RtlULongToInt8 function
- RtlULongToIntPtr function
- RtlULongToLong function
- RtlULongToLongPtr function
- RtlULongToShort function
- RtlULongToUChar function
- RtlULongToUInt function
- RtlULongToUInt8 function
- RtlULongToUIntPtr function
- RtlULongToUShort function
- RtlUShortAdd function
- RtlUShortMult function
- RtlUShortSub function
- RtlUShortToChar function
- RtlUShortToInt8 function
- RtlUShortToShort function
- RtlUShortToUChar function
- RtlUShortToUInt8 function

Ntpoapi.h

- POWER_ACTION enumeration
- POWER_INFORMATION_LEVEL enumeration
- POWER_MONITOR_REQUEST_REASON enumeration
- COUNTED_REASON_CONTEXT structure
- DEVICE_POWER_STATE enumeration
- MONITOR_DISPLAY_STATE enumeration

PO_SPR_ACTIVE_SESSION_DATA structure
POWER_PLATFORM_INFORMATION structure
POWER_REQUEST_TYPE enumeration
POWER_STATE union
POWER_STATE_TYPE enumeration
SYSTEM_POWER_STATE enumeration
SYSTEM_POWER_STATE_CONTEXT structure
CM_POWER_DATA structure
NtPowerInformation function
Ntstrsafe.h
RtlStringCbCatA function
RtlStringCbCatExA function
RtlStringCbCatExW function
RtlStringCbCatNA function
RtlStringCbCatNExA function
RtlStringCbCatNExW function
RtlStringCbCatNW function
RtlStringCbCatW function
RtlStringCbCopyA function
RtlStringCbCopyExA function
RtlStringCbCopyExW function
RtlStringCbCopyNA function
RtlStringCbCopyNExA function
RtlStringCbCopyNExW function
RtlStringCbCopyNW function
RtlStringCbCopyUnicodeString function
RtlStringCbCopyUnicodeStringEx function
RtlStringCbCopyW function
RtlStringCbLengthA function
RtlStringCbLengthW function
RtlStringCbPrintfA function
RtlStringCbPrintfExA function

RtlStringCbPrintfExW function

RtlStringCbPrintfW function

RtlStringCbVPrintfA function

RtlStringCbVPrintfExA function

RtlStringCbVPrintfExW function

RtlStringCbVPrintfW function

RtlStringCchCatA function

RtlStringCchCatExA function

RtlStringCchCatExW function

RtlStringCchCatNA function

RtlStringCchCatNExA function

RtlStringCchCatNExW function

RtlStringCchCatNW function

RtlStringCchCatW function

RtlStringCchCopyA function

RtlStringCchCopyExA function

RtlStringCchCopyExW function

RtlStringCchCopyNA function

RtlStringCchCopyNExA function

RtlStringCchCopyNExW function

RtlStringCchCopyNW function

RtlStringCchCopyUnicodeString function

RtlStringCchCopyUnicodeStringEx function

RtlStringCchCopyW function

RtlStringCchLengthA function

RtlStringCchLengthW function

RtlStringCchPrintfA function

RtlStringCchPrintfExA function

RtlStringCchPrintfExW function

RtlStringCchPrintfW function

RtlStringCchVPrintfA function

RtlStringCchVPrintfExA function

RtlStringCchVPrintfExW function

RtlStringCchVPrintfW function

RtlUnalignedStringCbLengthW function

RtlUnalignedStringCchLengthW function

RtlUnicodeStringCat function

RtlUnicodeStringCatEx function

RtlUnicodeStringCatString function

RtlUnicodeStringCatStringEx function

RtlUnicodeStringCbCatN function

RtlUnicodeStringCbCatNEx function

RtlUnicodeStringCbCatStringN function

RtlUnicodeStringCbCatStringNEx function

RtlUnicodeStringCbCopyN function

RtlUnicodeStringCbCopyNEx function

RtlUnicodeStringCbCopyStringN function

RtlUnicodeStringCbCopyStringNEx function

RtlUnicodeStringCchCatN function

RtlUnicodeStringCchCatNEx function

RtlUnicodeStringCchCatStringN function

RtlUnicodeStringCchCatStringNEx function

RtlUnicodeStringCchCopyN function

RtlUnicodeStringCchCopyNEx function

RtlUnicodeStringCchCopyStringN function

RtlUnicodeStringCchCopyStringNEx function

RtlUnicodeStringCopy function

RtlUnicodeStringCopyEx function

RtlUnicodeStringCopyString function

RtlUnicodeStringCopyStringEx function

RtlUnicodeStringInit function

RtlUnicodeStringInitEx function

RtlUnicodeStringPrintf function

RtlUnicodeStringPrintfEx function

RtlUnicodeStringValidate function
RtlUnicodeStringValidateEx function
RtlUnicodeStringVPrintf function
RtlUnicodeStringVPrintfEx function
Pcivirt.h
Pep_x.h
PEP_PROCESSOR_IDLE_TYPE enumeration
PEP_PROCESSOR_IDLE_CANCEL_CODE enumeration
GPIO_PIN_CONFIG_TYPE enumeration
GPIO_PIN_IORESTRICTION_TYPE enumeration
PEP_ABANDON_DEVICE structure
PEP_ACPI_ABANDON_DEVICE structure
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure
PEP_ACPI_EVALUATE_CONTROL_METHOD structure
PEP_ACPI_EXTENDED_ADDRESS structure
PEP_ACPI_GPIO_RESOURCE structure
PEP_ACPI_INTERRUPT_RESOURCE structure
PEP_ACPI_IO_MEMORY_RESOURCE structure
PEP_ACPI_OBJECT_NAME union
PEP_ACPI_OBJECT_NAME_WITH_TYPE structure
PEP_ACPI_OBJECT_TYPE enumeration
PEP_ACPI_PREPARE_DEVICE structure
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure
PEP_ACPI_QUERY_OBJECT_INFORMATION structure
PEP_ACPI_REGISTER_DEVICE structure
PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure
PEP_ACPI_RESOURCE union
PEP_ACPI_RESOURCE_FLAGS union
PEP_ACPI_RESOURCE_TYPE enumeration
PEP_ACPI_SPB_I2C_RESOURCE structure
PEP_ACPI_SPB_RESOURCE structure
PEP_ACPI_SPB_SPI_RESOURCE structure

PEP_ACPI_SPB_UART_RESOURCE structure

PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure

PEP_ACPI_UNREGISTER_DEVICE structure

PEP_COMPONENT_ACTIVE structure

PEP_COMPONENT_PERF_INFO structure

PEP_COMPONENT_PERF_SET structure

PEP_COMPONENT_PERF_STATE_REQUEST structure

PEP_COMPONENT_PLATFORM_CONSTRAINTS structure

PEP_COMPONENT_V2 structure

PEP_COORDINATED_DEPENDENCY_OPTION structure

PEP_COORDINATED_IDLE_STATE structure

PEP_CRASHDUMP_INFORMATION structure

PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure

PEP_DEVICE_ACCEPTANCE_TYPE enumeration

PEP_DEVICE_PLATFORM_CONSTRAINTS structure

PEP_DEVICE_POWER_STATE structure

PEP_DEVICE_REGISTER_V2 structure

PEP_DEVICE_STARTED structure

PEP_INFORMATION structure

PEP_KERNEL_INFORMATION_STRUCT_V1 structure

PEP_KERNEL_INFORMATION_STRUCT_V2 structure

PEP_KERNEL_INFORMATION_STRUCT_V3 structure

PEP_LOW_POWER_EPOCH structure

PEP_NOTIFY_COMPONENT_IDLE_STATE structure

PEP_PERF_STATE structure

PEP_PERF_STATE_TYPE enumeration

PEP_PERF_STATE_UNIT enumeration

PEP_PLATFORM_IDLE_STATE structure

PEP_PLATFORM_IDLE_STATE_UPDATE structure

PEP_POWER_CONTROL_COMPLETE structure

PEP_POWER_CONTROL_REQUEST structure

PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure

PEP_PPM_CST_STATE structure

PEP_PPM_CST_STATES structure

PEP_PPM_ENTER_SYSTEM_STATE structure

PEP_PPM_FEEDBACK_READ structure

PEP_PPM_IDLE_CANCEL structure

PEP_PPM_IDLE_COMPLETE structure

PEP_PPM_IDLE_COMPLETE_V2 structure

PEP_PPM_IDLE_EXECUTE structure

PEP_PPM_IDLE_EXECUTE_V2 structure

PEP_PPM_IDLE_SELECT structure

PEP_PPM_INITIATE_WAKE structure

PEP_PPM_IS_PROCESSOR_HALTED structure

PEP_PPM_LPI_COMPLETE structure

PEP_PPM_PARK_MASK structure

PEP_PPM_PARK_SELECTION structure

PEP_PPM_PARK_SELECTION_V2 structure

PEP_PPM_PERF_CHECK_COMPLETE structure

PEP_PPM_PERF_SET structure

PEP_PPM_PERF_SET_STATE structure

PEP_PPM_PLATFORM_STATE_RESIDENCIES structure

PEP_PPM_PLATFORM_STATE_RESIDENCY structure

PEP_PPM_QUERY_CAPABILITIES structure

PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure

PEP_PPM_QUERY_COORDINATED_STATES structure

PEP_PPM_QUERY_DISCRETE_PERF_STATES structure

PEP_PPM_QUERY_DOMAIN_INFO structure

PEP_PPM_QUERY_FEEDBACK_COUNTERS structure

PEP_PPM_QUERY_IDLE_STATES structure

PEP_PPM_QUERY_IDLE_STATES_V2 structure

PEP_PPM_QUERY_LP_SETTINGS structure

PEP_PPM_QUERY_PERF_CAPABILITIES structure

PEP_PPM_PERF_CONSTRAINTS structure

PEP_PPM_QUERY_PLATFORM_STATE structure

PEP_PPM_QUERY_PLATFORM_STATES structure

PEP_PPM_QUERY_STATE_NAME structure

PEP_PPM_QUERY_VETO_REASON structure

PEP_PPM_QUERY_VETO_REASONS structure

PEP_PPM_RESUME_FROM_SYSTEM_STATE structure

PEP_PPM_TEST_IDLE_STATE structure

PEP_PREPARE_DEVICE structure

PEP_PROCESSOR_FEEDBACK_COUNTER structure

PEP_PROCESSOR_IDLE_CONSTRAINTS structure

PEP_PROCESSOR_IDLE_DEPENDENCY structure

PEP_PROCESSOR_IDLE_STATE structure

PEP_PROCESSOR_IDLE_STATE_UPDATE structure

PEP_PROCESSOR_IDLE_STATE_V2 structure

PEP_PROCESSOR_PARK_PREFERENCE structure

PEP_PROCESSOR_PARK_STATE structure

PEP_PROCESSOR_PERF_STATE structure

PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure

PEP_QUERY_COMPONENT_PERF_SET structure

PEP_QUERY_COMPONENT_PERF_SET_NAME structure

PEP_QUERY_COMPONENT_PERF_STATES structure

PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure

PEP_QUERY_SOC_SUBSYSTEM structure

PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure

PEP_QUERY_SOC_SUBSYSTEM_COUNT structure

PEP_QUERY_SOC_SUBSYSTEM_METADATA structure

PEP_REGISTER_COMPONENT_PERF_STATES structure

PEP_REGISTER_CRASHDUMP_DEVICE structure

PEP_REGISTER_DEBUGGER structure

PEP_REGISTER_DEVICE_V2 structure

PEP_REQUEST_COMPONENT_PERF_STATE structure

PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure

PEP_SOC_SUBSYSTEM_METADATA structure

PEP_SYSTEM_LATENCY structure

PEP_UNMASKED_INTERRUPT_FLAGS union

PEP_UNMASKED_INTERRUPT_INFORMATION structure

PEP_UNREGISTER_DEVICE structure

PEP_WORK structure

PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure

PEP_WORK_ACPI_NOTIFY structure

PEP_WORK_ACTIVE_COMPLETE structure

PEP_WORK_COMPLETE_IDLE_STATE structure

PEP_WORK_COMPLETE_PERF_STATE structure

PEP_WORK_DEVICE_IDLE structure

PEP_WORK_DEVICE_POWER structure

PEP_WORK_IDLE_STATE structure

PEP_WORK_INFORMATION structure

PEP_WORK_POWER_CONTROL structure

PEP_WORK_TYPE enumeration

PO_FX_CORE_DEVICE structure

PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function

PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function

PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function

PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function

PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function

PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function

PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function

PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function

PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function

PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function

PEPCALLBACKNOTIFYACPI callback function

PEPCALLBACKNOTIFYDPM callback function

PEPCALLBACKNOTIFYPPM callback function

PEPCALLBACKPOWERONCRASHDUMPDEVICE callback function

POFXCALLBACKCRITICALRESOURCE callback function
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS callback function
POFXCALLBACKPLATFORMIDLEVETO callback function
POFXCALLBACKPROCESSORHALT callback function
POFXCALLBACKPROCESSORIDLEVETO callback function
POFXCALLBACKREQUESTCOMMON callback function
POFXCALLBACKREQUESTINTERRUPT callback function
POFXCALLBACKREQUESTWORKER callback function
POFXCALLBACKUPDATEPLATFORMIDLESTATE callback function
POFXCALLBACKUPDATEPROCESSORIDLESTATE callback function
PoFxRegisterCoreDevice function
PoFxRegisterPlugin function
PoFxRegisterPluginEx function
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK callback function
Pepevents.h
Pepfx.h
GPIO_PIN_CONFIG_TYPE enumeration
GPIO_PIN_IORESTRICTION_TYPE enumeration
PEP_ABANDON_DEVICE structure
PEP_ACPI_ABANDON_DEVICE structure
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure
PEP_ACPI_EVALUATE_CONTROL_METHOD structure
PEP_ACPI_EXTENDED_ADDRESS structure
PEP_ACPI_GPIO_RESOURCE structure
PEP_ACPI_INTERRUPT_RESOURCE structure
PEP_ACPI_IO_MEMORY_RESOURCE structure
PEP_ACPI_OBJECT_NAME union
PEP_ACPI_OBJECT_NAME_WITH_TYPE structure
PEP_ACPI_OBJECT_TYPE enumeration
PEP_ACPI_PREPARE_DEVICE structure
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure
PEP_ACPI_QUERY_OBJECT_INFORMATION structure

PEP_ACPI_REGISTER_DEVICE structure

PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure

PEP_ACPI_RESOURCE union

PEP_ACPI_RESOURCE_FLAGS union

PEP_ACPI_RESOURCE_TYPE enumeration

PEP_ACPI_SPB_I2C_RESOURCE structure

PEP_ACPI_SPB_RESOURCE structure

PEP_ACPI_SPB_SPI_RESOURCE structure

PEP_ACPI_SPB_UART_RESOURCE structure

PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure

PEP_ACPI_UNREGISTER_DEVICE structure

PEP_COMPONENT_PERF_INFO structure

PEP_COMPONENT_PERF_SET structure

PEP_COMPONENT_PERF_STATE_REQUEST structure

PEP_COMPONENT_PLATFORM_CONSTRAINTS structure

PEP_COMPONENT_V2 structure

PEP_COORDINATED_DEPENDENCY_OPTION structure

PEP_COORDINATED_IDLE_STATE structure

PEP_CRASHDUMP_INFORMATION structure

PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure

PEP_DEVICE_ACCEPTANCE_TYPE enumeration

PEP_DEVICE_PLATFORM_CONSTRAINTS structure

PEP_DEVICE_POWER_STATE structure

PEP_DEVICE_REGISTER_V2 structure

PEP_DEVICE_STARTED structure

PEP_INFORMATION structure

PEP_KERNEL_INFORMATION_STRUCT_V3 structure

PEP_LOW_POWER_EPOCH structure

PEP_NOTIFY_COMPONENT_IDLE_STATE structure

PEP_PERF_STATE structure

PEP_PERF_STATE_TYPE enumeration

PEP_PERF_STATE_UNIT enumeration

PEP_PLATFORM_IDLE_STATE structure

PEP_PLATFORM_IDLE_STATE_UPDATE structure

PEP_POWER_CONTROL_COMPLETE structure

PEP_POWER_CONTROL_REQUEST structure

PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure

PEP_PPM_CST_STATE structure

PEP_PPM_CST_STATES structure

PEP_PPM_ENTER_SYSTEM_STATE structure

PEP_PPM_FEEDBACK_READ structure

PEP_PPM_IDLE_COMPLETE structure

PEP_PPM_IDLE_COMPLETE_V2 structure

PEP_PPM_IDLE_EXECUTE structure

PEP_PPM_IDLE_EXECUTE_V2 structure

PEP_PPM_INITIATE_WAKE structure

PEP_PPM_IS_PROCESSOR_HALTED structure

PEP_PPM_PARK_MASK structure

PEP_PPM_PARK_SELECTION structure

PEP_PPM_PARK_SELECTION_V2 structure

PEP_PPM_PERF_CHECK_COMPLETE structure

PEP_PPM_PERF_SET structure

PEP_PPM_PERF_SET_STATE structure

PEP_PPM_PLATFORM_STATE_RESIDENCIES structure

PEP_PPM_PLATFORM_STATE_RESIDENCY structure

PEP_PPM_QUERY_CAPABILITIES structure

PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure

PEP_PPM_QUERY_COORDINATED_STATES structure

PEP_PPM_QUERY_DISCRETE_PERF_STATES structure

PEP_PPM_QUERY_DOMAIN_INFO structure

PEP_PPM_QUERY_FEEDBACK_COUNTERS structure

PEP_PPM_QUERY_IDLE_STATES_V2 structure

PEP_PPM_QUERY_PERF_CAPABILITIES structure

PEP_PPM_PERF_CONSTRAINTS structure

PEP_PPM_QUERY_PLATFORM_STATE structure

PEP_PPM_QUERY_PLATFORM_STATES structure

PEP_PPM_QUERY_STATE_NAME structure

PEP_PPM_QUERY_VETO_REASON structure

PEP_PPM_QUERY_VETO_REASONS structure

PEP_PPM_RESUME_FROM_SYSTEM_STATE structure

PEP_PPM_TEST_IDLE_STATE structure

PEP_PREPARE_DEVICE structure

PEP_PROCESSOR_FEEDBACK_COUNTER structure

PEP_PROCESSOR_IDLE_DEPENDENCY structure

PEP_PROCESSOR_IDLE_STATE_UPDATE structure

PEP_PROCESSOR_IDLE_STATE_V2 structure

PEP_PROCESSOR_PARK_PREFERENCE structure

PEP_PROCESSOR_PARK_STATE structure

PEP_PROCESSOR_PERF_STATE structure

PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure

PEP_QUERY_COMPONENT_PERF_SET structure

PEP_QUERY_COMPONENT_PERF_SET_NAME structure

PEP_QUERY_COMPONENT_PERF_STATES structure

PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure

PEP_QUERY_SOC_SUBSYSTEM structure

PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure

PEP_QUERY_SOC_SUBSYSTEM_COUNT structure

PEP_QUERY_SOC_SUBSYSTEM_METADATA structure

PEP_REGISTER_COMPONENT_PERF_STATES structure

PEP_REGISTER_CRASHDUMP_DEVICE structure

PEP_REGISTER_DEBUGGER structure

PEP_REGISTER_DEVICE_V2 structure

PEP_REQUEST_COMPONENT_PERF_STATE structure

PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure

PEP_SOC_SUBSYSTEM_METADATA structure

PEP_SYSTEM_LATENCY structure

PEP_UNMASKED_INTERRUPT_FLAGS union
PEP_UNMASKED_INTERRUPT_INFORMATION structure
PEP_UNREGISTER_DEVICE structure
PEP_WORK structure
PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure
PEP_WORK_ACPI_NOTIFY structure
PEP_WORK_COMPLETE_IDLE_STATE structure
PEP_WORK_COMPLETE_PERF_STATE structure
PEP_WORK_INFORMATION structure
PEP_WORK_POWER_CONTROL structure
PEP_WORK_TYPE enumeration
PO_FX_CORE_DEVICE structure
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function
PEPCALLBACKNOTIFYACPI callback function
PEPCALLBACKNOTIFYDPM callback function
PEPCALLBACKNOTIFYPPM callback function
PEPCALLBACKPOWERONCRASHDUMPDEVICE callback function
POFXCALLBACKCRITICALRESOURCE callback function
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS callback function
POFXCALLBACKPLATFORMIDLEVETO callback function
POFXCALLBACKPROCESSORHALT callback function
POFXCALLBACKPROCESSORIDLEVETO callback function
POFXCALLBACKREQUESTCOMMON callback function

POFXCALLBACKREQUESTINTERRUPT callback function
POFXCALLBACKREQUESTWORKER callback function
POFXCALLBACKUPDATEPLATFORMIDLESTATE callback function
POFXCALLBACKUPDATEPROCESSORIDLESTATE callback function
PoFxRegisterCoreDevice function
PoFxRegisterPlugin function
PoFxRegisterPluginEx function
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK callback function
Procgrp.h
WdmlibProcgrpInitialize function
Vpci.h
Wdm.h
DMA_COMPLETION_STATUS enumeration
POWER_INFORMATION_LEVEL enumeration
DEVICE_REGISTRY_PROPERTY enumeration
BOOTDISK_INFORMATION structure
BOOTDISK_INFORMATION_EX structure
BOUND_CALLBACK_STATUS enumeration
BUS_INTERFACE_STANDARD structure
CLFS_CONTEXT_MODE enumeration
CLFS_LOG_NAME_INFORMATION structure
CLFS_MGMT_CLIENT_REGISTRATION structure
CLFS_MGMT_POLICY structure
CLFS_MGMT_POLICY_TYPE enumeration
CLFS_STREAM_ID_INFORMATION structure
CLS_CONTAINER_INFORMATION structure
CLS_INFORMATION structure
CLS_IO_STATISTICS structure
CLS_IO_STATISTICS_HEADER structure
CLS_LOG_INFORMATION_CLASS enumeration
CLS_LSN structure
CLS_SCAN_CONTEXT structure

CLS_WRITE_ENTRY structure

CM_EISA_FUNCTION_INFORMATION structure

CM_EISA_SLOT_INFORMATION structure

CM_FLOPPY_DEVICE_DATA structure

CM_FULL_RESOURCE_DESCRIPTOR structure

CM_INT13_DRIVE_PARAMETER structure

CM_KEYBOARD_DEVICE_DATA structure

CM_MCA_POS_DATA structure

CM_PARTIAL_RESOURCE_DESCRIPTOR structure

CM_PARTIAL_RESOURCE_LIST structure

CM_RESOURCE_LIST structure

CM_SCSI_DEVICE_DATA structure

CM_SERIAL_DEVICE_DATA structure

COUNTED_REASON_CONTEXT structure

D3COLD_LAST_TRANSITION_STATUS enumeration

D3COLD_SUPPORT_INTERFACE structure

DEVICE_CAPABILITIES structure

DEVICE_DESCRIPTION structure

DEVICE_INSTALL_STATE enumeration

DEVICE_INTERFACE_CHANGE_NOTIFICATION structure

DEVICE_OBJECT structure

DEVICE_POWER_STATE enumeration

DEVICE_REMOVAL_POLICY enumeration

DEVICE_RESET_INTERFACE_STANDARD structure

DEVICE_RESET_TYPE enumeration

DEVICE_WAKE_DEPTH enumeration

DMA_ADAPTER structure

DMA_ADAPTER_INFO structure

DMA_ADAPTER_INFO_V1 structure

DMA_OPERATIONS structure

DMA_TRANSFER_INFO structure

DMA_TRANSFER_INFO_V1 structure

DRIVER_OBJECT structure
ENLISTMENT_BASIC_INFORMATION structure
ENLISTMENT_INFORMATION_CLASS enumeration
EXT_DELETE_PARAMETERS structure
EXT_SET_PARAMETERS structure
FILE_BASIC_INFORMATION structure
FILE_FS_DEVICE_INFORMATION structure
FILE_FULL_EA_INFORMATION structure
FILE_IO_PRIORITY_HINT_INFORMATION structure
FILE_IS_REMOTE_DEVICE_INFORMATION structure
FILE_NETWORK_OPEN_INFORMATION structure
FILE_OBJECT structure
FILE_POSITION_INFORMATION structure
FILE_STANDARD_INFORMATION structure
FILE_STANDARD_INFORMATION_EX structure
FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS structure
GENERIC_MAPPING structure
HWPROFILE_CHANGE_NOTIFICATION structure
IMAGE_POLICY_ENTRY structure
IMAGE_POLICY_ENTRY_TYPE enumeration
IMAGE_POLICY_ID enumeration
IMAGE_POLICY_METADATA structure
INTERFACE structure
INTERFACE_TYPE enumeration
IO_ACCESS_MODE enumeration
IO_ACCESS_TYPE enumeration
IO_ALLOCATION_ACTION enumeration
IO_CONNECT_INTERRUPT_PARAMETERS structure
IO_CONTAINER_INFORMATION_CLASS enumeration
IO_CONTAINER_NOTIFICATION_CLASS enumeration
IO_DISCONNECT_INTERRUPT_PARAMETERS structure
IO_ERROR_LOG_PACKET structure

IO_INTERRUPT_MESSAGE_INFO structure

IO_INTERRUPT_MESSAGE_INFO_ENTRY structure

IO_PAGING_PRIORITY enumeration

IO_PRIORITY_HINT enumeration

IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure

IO_RESOURCE_DESCRIPTOR structure

IO_RESOURCE_LIST structure

IO_RESOURCE_REQUIREMENTS_LIST structure

IO_SECURITY_CONTEXT structure

IO_SESSION_CONNECT_INFO structure

IO_SESSION_EVENT enumeration

IO_SESSION_STATE enumeration

IO_SESSION_STATE_INFORMATION structure

IO_SESSION_STATE_NOTIFICATION structure

IO_STACK_LOCATION structure

IO_STATUS_BLOCK structure

IRP structure

IRQ_DEVICE_POLICY enumeration

IRQ_PRIORITY enumeration

KBUGCHECK_ADD_PAGES structure

KBUGCHECK_CALLBACK_REASON enumeration

KBUGCHECK_DUMP_IO structure

KBUGCHECK_DUMP_IO_TYPE enumeration

KBUGCHECK_SECONDARY_DUMP_DATA structure

KDPC_WATCHDOG_INFORMATION structure

KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure

KEY_BASIC_INFORMATION structure

KEY_FULL_INFORMATION structure

KEY_INFORMATION_CLASS enumeration

KEY_NODE_INFORMATION structure

KEY_SET_INFORMATION_CLASS enumeration

KEY_VALUE_BASIC_INFORMATION structure

KEY_VALUE_ENTRY structure

KEY_VALUE_FULL_INFORMATION structure

KEY_VALUE_INFORMATION_CLASS enumeration

KEY_VALUE_PARTIAL_INFORMATION structure

KEY_WRITE_TIME_INFORMATION structure

KINTERRUPT_MODE enumeration

KINTERRUPT_POLARITY enumeration

KTMOBJECT_CURSOR structure

KTMOBJECT_TYPE enumeration

LINK_SHARE_ACCESS structure

MDL structure

MEMORY_CACHING_TYPE enumeration

MM_PHYSICAL_ADDRESS_LIST structure

MONITOR_DISPLAY_STATE enumeration

OB_CALLBACK_REGISTRATION structure

OB_OPERATION_REGISTRATION structure

OB_POST_CREATE_HANDLE_INFORMATION structure

OB_POST_DUPLICATE_HANDLE_INFORMATION structure

OB_POST_OPERATION_INFORMATION structure

OB_POST_OPERATION_PARAMETERS union

OB_PRE_CREATE_HANDLE_INFORMATION structure

OB_PRE_DUPLICATE_HANDLE_INFORMATION structure

OB_PRE_OPERATION_INFORMATION structure

OB_PRE_OPERATION_PARAMETERS union

OSVERSIONINFOEXW structure

OSVERSIONINFOW structure

PCI_COMMON_CONFIG structure

PCI_MSIX_TABLE_CONFIG_INTERFACE structure

PCI_SLOT_NUMBER structure

PLUGPLAY_NOTIFICATION_HEADER structure

PNP_BUS_INFORMATION structure

PO_FX_COMPONENT_IDLE_STATE structure

PO_FX_COMPONENT_PERF_INFO structure

PO_FX_COMPONENT_PERF_SET structure

PO_FX_COMPONENT_V1 structure

PO_FX_COMPONENT_V2 structure

PO_FX_DEVICE_V1 structure

PO_FX_DEVICE_V2 structure

PO_FX_PERF_STATE structure

PO_FX_PERF_STATE_CHANGE structure

PO_FX_PERF_STATE_TYPE enumeration

PO_FX_PERF_STATE_UNIT enumeration

POOL_TYPE enumeration

POWER_PLATFORM_INFORMATION structure

POWER_REQUEST_TYPE enumeration

POWER_STATE union

POWER_STATE_TYPE enumeration

PRIVILEGE_SET structure

REENUMERATE_SELF_INTERFACE_STANDARD structure

REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION structure

REG_CREATE_KEY_INFORMATION structure

REG_CREATE_KEY_INFORMATION_V1 structure

REG_DELETE_KEY_INFORMATION structure

REG_DELETE_VALUE_KEY_INFORMATION structure

REG_ENUMERATE_KEY_INFORMATION structure

REG_ENUMERATE_VALUE_KEY_INFORMATION structure

REG_KEY_HANDLE_CLOSE_INFORMATION structure

REG_LOAD_KEY_INFORMATION structure

REG_NOTIFY_CLASS enumeration

REG_POST_CREATE_KEY_INFORMATION structure

REG_POST_OPERATION_INFORMATION structure

REG_PRE_CREATE_KEY_INFORMATION structure

REG_QUERY_KEY_INFORMATION structure

REG_QUERY_KEY_NAME structure

REG_QUERY_KEY_SECURITY_INFORMATION structure

REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure

REG_QUERY_VALUE_KEY_INFORMATION structure

REG_RENAME_KEY_INFORMATION structure

REG_REPLACE_KEY_INFORMATION structure

REG_RESTORE_KEY_INFORMATION structure

REG_SAVE_KEY_INFORMATION structure

REG_SET_INFORMATION_KEY_INFORMATION structure

REG_SET_KEY_SECURITY_INFORMATION structure

REG_SET_VALUE_KEY_INFORMATION structure

REG_UNLOAD_KEY_INFORMATION structure

RESOURCEMANAGER_BASIC_INFORMATION structure

RESOURCEMANAGER_COMPLETION_INFORMATION structure

RESOURCEMANAGER_INFORMATION_CLASS enumeration

SCATTER_GATHER_LIST structure

SDEV_IDENTIFIER_INTERFACE structure

SECTION_OBJECT_POINTERS structure

SLIST_ENTRY structure

SYSTEM_POWER_STATE enumeration

SYSTEM_POWER_STATE_CONTEXT structure

TARGET_DEVICE_CUSTOM_NOTIFICATION structure

TARGET_DEVICE_REMOVAL_NOTIFICATION structure

TRACE_INFORMATION_CLASS enumeration

TRANSACTION_BASIC_INFORMATION structure

TRANSACTION_ENLISTMENT_PAIR structure

TRANSACTION_ENLISTMENTS_INFORMATION structure

TRANSACTION_INFORMATION_CLASS enumeration

TRANSACTION_OUTCOME enumeration

TRANSACTION_PROPERTIES_INFORMATION structure

TRANSACTION_STATE enumeration

TRANSACTIONMANAGER_BASIC_INFORMATION structure

TRANSACTIONMANAGER_INFORMATION_CLASS enumeration

TRANSACTIONMANAGER_LOG_INFORMATION structure
TRANSACTIONMANAGER_LOGPATH_INFORMATION structure
TRANSACTIONMANAGER_RECOVERY_INFORMATION structure
WORK_QUEUE_TYPE enumeration
ALLOCATE_FUNCTION_EX callback function
AppendTailList function
BOUND_CALLBACK callback function
ClfsAddLogContainer function
ClfsAddLogContainerSet function
ClfsAdvanceLogBase function
ClfsAlignReservedLog function
ClfsAllocReservedLog function
ClfsCloseAndResetLogFile function
ClfsCloseLogFileObject function
ClfsCreateLogFile function
ClfsCreateMarshallingArea function
ClfsCreateScanContext function
ClfsDeleteLogByPointer function
ClfsDeleteLogFile function
ClfsDeleteMarshallingArea function
ClfsFlushBuffers function
ClfsFlushToLsn function
ClfsGetContainerName function
ClfsGetIoStatistics function
ClfsLsnBlockOffset function
ClfsLsnContainer function
ClfsLsnCreate function
ClfsLsnEqual function
ClfsLsnGreater function
ClfsLsnLess function
ClfsLsnNull function
ClfsLsnRecordSequence function

ClfsMgmtDeregisterManagedClient function

ClfsMgmtHandleLogFileFull function

ClfsMgmtInstallPolicy function

ClfsMgmtQueryPolicy function

ClfsMgmtRegisterManagedClient function

ClfsMgmtRemovePolicy function

ClfsMgmtSetLogFileSize function

ClfsMgmtSetLogFileSizeAsClient function

ClfsMgmtTailAdvanceFailure function

ClfsQueryLogFileInformation function

ClfsReadLogRecord function

ClfsReadNextLogRecord function

ClfsReadPreviousRestartArea function

ClfsReadRestartArea function

ClfsRemoveLogContainer function

ClfsRemoveLogContainerSet function

ClfsReserveAndAppendLog function

ClfsReserveAndAppendLogAligned function

ClfsScanLogContainers function

ClfsSetArchiveTail function

ClfsSetEndOfLog function

ClfsSetLogFileInformation function

ClfsTerminateReadLog function

ClfsWriteRestartArea function

CM_POWER_DATA structure

CmCallbackGetKeyObjectID function

CmCallbackGetKeyObjectIDEx function

CmCallbackReleaseKeyObjectIDEx function

CmGetBoundTransaction function

CmGetCallbackVersion function

CmRegisterCallback function

CmRegisterCallbackEx function

CmSetCallbackObjectContext function
CmUnRegisterCallback function
DMA_COMPLETION_ROUTINE callback function
DRIVER_ADD_DEVICE callback function
DRIVER_CANCEL callback function
DRIVER_CONTROL callback function
DRIVER_DISPATCH callback function
DRIVER_INITIALIZE callback function
DRIVER_LIST_CONTROL callback function
DRIVER_STARTIO callback function
DRIVER_UNLOAD callback function
EX_CALLBACK_FUNCTION callback function
ExAcquireResourceExclusiveLite function
ExAcquireResourceSharedLite function
ExAcquireRundownProtection function
ExAcquireRundownProtectionEx function
ExAcquireSharedStarveExclusive function
ExAcquireSharedWaitForExclusive function
ExAllocateFromLookasideListEx function
ExAllocateFromNPagedLookasideList function
ExAllocateFromPagedLookasideList function
ExAllocatePool function
ExAllocatePoolWithQuota function
ExAllocatePoolWithQuotaTag function
ExAllocatePoolWithTag function
ExAllocatePoolWithTagPriority function
ExAllocateTimer function
ExCancelTimer function
ExConvertExclusiveToSharedLite function
ExCreateCallback function
ExDeleteLookasideListEx function
ExDeleteNPagedLookasideList function

ExDeletePagedLookasideList function

ExDeleteResourceLite function

ExDeleteTimer function

ExFlushLookasideListEx function

ExFreePool function

ExFreePoolWithTag function

ExFreeToLookasideListEx function

ExFreeToNPagedLookasideList function

ExFreeToPagedLookasideList function

ExGetExclusiveWaiterCount function

ExGetFirmwareEnvironmentVariable function

ExGetFirmwareType function

ExGetPreviousMode function

ExGetSharedWaiterCount function

ExInitializeDeleteTimerParameters function

ExInitializeFastMutex function

ExInitializeLookasideListEx function

ExInitializeNPagedLookasideList function

ExInitializePagedLookasideList function

ExInitializeResourceLite function

ExInitializeRundownProtection function

ExInitializeSetTimerParameters function

ExInterlockedAddLargeInteger function

ExInterlockedAddLargeStatistic macro

ExInterlockedAddUlong function

ExInterlockedCompareExchange64 macro

ExInterlockedFlushSList macro

ExInterlockedInsertHeadList function

ExInterlockedInsertTailList function

ExInterlockedPopEntryList function

ExInterlockedPopEntrySList macro

ExInterlockedPushEntryList function

ExInterlockedPushEntrySList macro

ExInterlockedRemoveHeadList function

ExIsProcessorFeaturePresent function

ExIsResourceAcquiredExclusiveLite function

ExIsResourceAcquiredSharedLite function

ExIsSoftBoot function

ExLocalTimeToSystemTime function

ExNotifyCallback function

ExQueryDepthSList function

ExQueryTimerResolution function

ExRaiseStatus function

ExRegisterCallback function

ExReinitializeResourceLite function

ExReInitializeRundownProtection function

ExReleaseResourceForThreadLite function

ExReleaseResourceLite function

ExReleaseRundownProtection function

ExReleaseRundownProtectionEx function

ExReleaseSpinLockExclusive function

ExReleaseSpinLockShared function

ExRundownCompleted function

ExSetFirmwareEnvironmentVariable function

ExSetResourceOwnerPointer function

ExSetResourceOwnerPointerEx function

ExSetTimer function

ExSetTimerResolution function

ExSystemTimeToLocalTime function

EXT_CALLBACK callback function

EXT_DELETE_CALLBACK callback function

ExTryConvertSharedSpinLockExclusive function

ExUnregisterCallback function

ExWaitForRundownProtectionRelease function

FIELD_OFFSET macro

FirstEntrySList function

FREE_FUNCTION_EX callback function

GET_D3COLD_CAPABILITY callback function

GET_D3COLD_LAST_TRANSITION_STATUS callback function

GET_DMA_ADAPTER callback function

GET_IDLE_WAKE_INFO callback function

GET_SDEV_IDENTIFIER callback function

GET_SET_DEVICE_DATA callback function

InitializeListHead function

InitializeSListHead function

InsertHeadList function

InsertTailList function

InterlockedAnd function

InterlockedCompareExchange function

InterlockedCompareExchangePointer function

InterlockedDecrement function

InterlockedExchange function

InterlockedExchangeAdd function

InterlockedExchangePointer function

InterlockedIncrement function

InterlockedOr function

InterlockedXor function

IO_COMPLETION_ROUTINE callback function

IO_CSQ_ACQUIRE_LOCK callback function

IO_CSQ_COMPLETE_CANCELED_IRP callback function

IO_CSQ_INSERT_IRP callback function

IO_CSQ_INSERT_IRP_EX callback function

IO_CSQ_PEEK_NEXT_IRP callback function

IO_CSQ_RELEASE_LOCK callback function

IO_CSQ_REMOVE_IRP callback function

IO_DPC_ROUTINE callback function

IO_SESSION_NOTIFICATION_FUNCTION callback function
IO_TIMER_ROUTINE callback function
IO_WORKITEM_ROUTINE callback function
IO_WORKITEM_ROUTINE_EX callback function
IoAcquireRemoveLock macro
IoAdjustPagingPathCount macro
IoAllocateDriverObjectExtension function
IoAllocateErrorLogEntry function
IoAllocateIrp function
IoAllocateMdl function
IoAllocateWorkItem function
IoAttachDevice function
IoAttachDeviceToDeviceStack function
IoBuildAsynchronousFsdRequest function
IoBuildDeviceIoControlRequest function
IoBuildPartialMdl function
IoBuildSynchronousFsdRequest function
IoCallDriver macro
IoCancelIrp function
IoCheckLinkShareAccess function
IoCheckShareAccess function
IoCheckShareAccessEx function
IoCompleteRequest macro
IoConnectInterrupt function
IoConnectInterruptEx function
IoCopyCurrentIrpStackLocationToNext function
IoCreateDevice function
IoCreateFile function
IoCreateNotificationEvent function
IoCreateSymbolicLink function
IoCreateSynchronizationEvent function
IoCreateSystemThread function

IoCreateUnprotectedSymbolicLink function
IoCsqInitialize function
IoCsqInitializeEx function
IoCsqInsertIrp function
IoCsqInsertIrpEx function
IoCsqRemoveIrp function
IoCsqRemoveNextIrp function
IoDeleteDevice function
IoDeleteSymbolicLink function
IoDetachDevice function
IoDisconnectInterrupt function
IoDisconnectInterruptEx function
IoForwardIrpSynchronously function
IoFreeErrorLogEntry function
IoFreeIrp function
IoFreeMdl function
IoFreeWorkItem function
IoGetAffinityInterrupt function
IoGetAttachedDeviceReference function
IoGetBootDiskInformation function
IoGetContainerInformation function
IoGetCurrentIrpStackLocation function
IoGetCurrentProcess function
IoGetDeviceInterfaceAlias function
IoGetDeviceInterfacePropertyData function
IoGetDeviceInterfaces function
IoGetDeviceNumaNode function
IoGetDeviceObjectPointer function
IoGetDeviceProperty function
IoGetDevicePropertyData function
IoGetDmaAdapter function
IoGetDriverObjectExtension function

IoGetFunctionCodeFromCtlCode macro

IoGetInitialStack function

IoGetIoPriorityHint function

IoGetNextIrpStackLocation function

IoGetRelatedDeviceObject function

IoGetRemainingStackSize function

IoGetStackLimits function

IoInitializeDpcRequest function

IoInitializeIrp function

IoInitializeRemoveLock macro

IoInitializeTimer function

IoInitializeWorkItem function

IoInvalidateDeviceRelations function

IoInvalidateDeviceState function

IoIs32bitProcess function

IoIsErrorUserInduced macro

IoIsWdmVersionAvailable function

IoMarkIrpPending function

IoOpenDeviceInterfaceRegistryKey function

IoOpenDeviceRegistryKey function

IoQueueWorkItem function

IoQueueWorkItemEx function

IoRegisterContainerNotification function

IoRegisterDeviceInterface function

IoRegisterLastChanceShutdownNotification function

IoRegisterPlugPlayNotification function

IoRegisterShutdownNotification function

IoReleaseRemoveLock macro

IoReleaseRemoveLockAndWait macro

IoRemoveLinkShareAccess function

IoRemoveShareAccess function

IoReportInterruptActive function

IoReportInterruptInactive function

IoReportTargetDeviceChange function

IoReportTargetDeviceChangeAsynchronous function

IoRequestDeviceEject function

IoRequestDpc function

IoReuseIrp function

IoSetCancelRoutine function

IoSetCompletionRoutine function

IoSetCompletionRoutineEx function

IoSetDeviceInterfacePropertyData function

IoSetDeviceInterfaceState function

IoSetDevicePropertyData function

IoSetIoPriorityHint function

IoSetLinkShareAccess function

IoSetNextIrpStackLocation function

IoSetShareAccess function

IoSetShareAccessEx function

IoSetStartIoAttributes function

IoSizeOfIrp macro

IoSizeofWorkItem function

IoStartNextPacket function

IoStartNextPacketByKey function

IoStartPacket function

IoStartTimer function

IoStopTimer function

IoUninitializeWorkItem function

IoUnregisterContainerNotification function

IoUnregisterPlugPlayNotification function

IoUnregisterPlugPlayNotificationEx function

IoUnregisterShutdownNotification function

IoUpdateLinkShareAccess function

IoUpdateShareAccess function

IoValidateDeviceIoControlAccess function

IoWithinStackLimits function

IoWMIAllocateInstanceIds function

IoWMIDeviceObjectToInstanceName function

IoWMIDeviceObjectToProviderId function

IoWMIExecuteMethod function

IoWMIHandleToInstanceName function

IoWMIOpenBlock function

IoWMIQueryAllData function

IoWMIQueryAllDataMultiple function

IoWMIQuerySingleInstance function

IoWMIQuerySingleInstanceMultiple function

IoWMIRegistrationControl function

IoWMISetNotificationCallback function

IoWMISetSingleInstance function

IoWMISetSingleItem function

IoWMISuggestInstanceName function

IoWMIWriteEvent function

IoWriteErrorLogEntry function

IsListEmpty function

KBUGCHECK_CALLBACK_ROUTINE callback function

KBUGCHECK_REASON_CALLBACK_ROUTINE callback function

KDEFERRED_ROUTINE callback function

KeAcquireSpinLock macro

KeAcquireSpinLockAtDpcLevel macro

KeAreAllApcsDisabled function

KeAreApcsDisabled function

KeBugCheckEx function

KeCancelTimer function

KeClearEvent function

KeConvertAuxiliaryCounterToPerformanceCounter function

KeConvertPerformanceCounterToAuxiliaryCounter function

KeDelayExecutionThread function

KeDeregisterBoundCallback function

KeDeregisterBugCheckCallback function

KeDeregisterBugCheckReasonCallback function

KeDeregisterNmiCallback function

KeDeregisterProcessorChangeCallback function

KeEnterCriticalRegion function

KeEnterGuardedRegion function

KeFlushIoBuffers function

KeFlushQueuedDpcs function

KefReleaseSpinLockFromDpcLevel function

KeGetCurrentIrql function

KeGetCurrentNodeNumber function

KeGetCurrentProcessorNumberEx function

KeGetCurrentThread function

KeGetProcessorIndexFromNumber function

KeGetProcessorNumberFromIndex function

KeGetRecommendedSharedDataAlignment function

KeInitializeCrashDumpHeader function

KeInitializeDeviceQueue function

KeInitializeDpc function

KeInitializeEvent function

KeInitializeGuardedMutex function

KeInitializeMutex function

KeInitializeSemaphore function

KeInitializeSpinLock function

KeInitializeThreadedDpc function

KeInitializeTimer function

KeInitializeTimerEx function

KeInsertByKeyDeviceQueue function

KeInsertDeviceQueue function

KeInsertQueueDpc function

KeIpiGenericCall function

KeLeaveCriticalRegion function

KeLeaveGuardedRegion function

KeLowerIrql function

KeMemoryBarrier function

KeQueryActiveGroupCount function

KeQueryActiveProcessorCount function

KeQueryActiveProcessorCountEx function

KeQueryActiveProcessors function

KeQueryAuxiliaryCounterFrequency function

KeQueryDpcWatchdogInformation function

KeQueryGroupAffinity function

KeQueryHighestNodeNumber function

KeQueryInterruptTime function

KeQueryInterruptTimePrecise function

KeQueryLogicalProcessorRelationship function

KeQueryMaximumGroupCount function

KeQueryMaximumProcessorCount function

KeQueryMaximumProcessorCountEx function

KeQueryNodeActiveAffinity function

KeQueryNodeMaximumProcessorCount function

KeQueryPerformanceCounter function

KeQueryPriorityThread function

KeQueryRuntimeThread function

KeQuerySystemTime function

KeQuerySystemTimePrecise function

KeQueryTickCount macro

KeQueryTimeIncrement function

KeQueryTotalCycleTimeThread function

KeQueryUnbiasedInterruptTime function

KeRaiseIrql macro

KeRaiseIrqlToDpcLevel function

KeReadStateEvent function

KeReadStateMutex function

KeReadStateSemaphore function

KeReadStateTimer function

KeRegisterBoundCallback function

KeRegisterBugCheckCallback function

KeRegisterBugCheckReasonCallback function

KeRegisterNmiCallback function

KeRegisterProcessorChangeCallback function

KeReleaseGuardedMutex function

KeReleaseGuardedMutexUnsafe function

KeReleaseInStackQueuedSpinLock function

KeReleaseInStackQueuedSpinLockForDpc function

KeReleaseInStackQueuedSpinLockFromDpcLevel function

KeReleaseInterruptSpinLock function

KeReleaseMutex function

KeReleaseSemaphore function

KeReleaseSpinLock macro

KeReleaseSpinLockForDpc function

KeReleaseSpinLockFromDpcLevel macro

KeRemoveByKeyDeviceQueue function

KeRemoveDeviceQueue function

KeRemoveEntryDeviceQueue function

KeRemoveQueueDpc function

KeResetEvent function

KeRestoreExtendedProcessorState function

KeRestoreFloatingPointState function

KeRevertToUserAffinityThreadEx function

KeRevertToUserGroupAffinityThread function

KeSaveExtendedProcessorState function

KeSaveFloatingPointState function

KeSetCoalescableTimer function

KeSetEvent function

KeSetImportanceDpc function

KeSetPriorityThread function

KeSetSystemAffinityThread function

KeSetSystemAffinityThreadEx function

KeSetSystemGroupAffinityThread function

KeSetTargetProcessorDpc function

KeSetTargetProcessorDpcEx function

KeSetTimer function

KeSetTimerEx function

KeStallExecutionProcessor function

KeSynchronizeExecution function

KeTestSpinLock function

KeTryToAcquireSpinLockAtDpcLevel function

KeWaitForMultipleObjects function

KeWaitForSingleObject function

KIPI_BROADCAST_WORKER callback function

KMESSAGE_SERVICE_ROUTINE callback function

KSERVICE_ROUTINE callback function

KSTART_ROUTINE callback function

KSYNCHRONIZE_ROUTINE callback function

MM_MDL_ROUTINE callback function

MmAdvanceMdl function

MmAllocateContiguousMemory function

MmAllocateContiguousMemorySpecifyCache function

MmAllocateContiguousMemorySpecifyCacheNode function

MmAllocateContiguousNodeMemory function

MmAllocateMappingAddress function

MmAllocateMdlForIoSpace function

MmAllocateNodePagesForMdlEx function

MmAllocatePagesForMdl function

MmAllocatePagesForMdlEx function

MmBuildMdlForNonPagedPool function

MmFreeContiguousMemory function

MmFreeContiguousMemorySpecifyCache function

MmFreeMappingAddress function

MmFreePagesFromMdl function

MmGetSystemAddressForMdl macro

MmGetSystemRoutineAddress function

MmIsDriverSuspectForVerifier function

MmIsDriverVerifying function

MmIsDriverVerifyingByAddress function

MmLockPagableCodeSection macro

MmLockPagableDataSection function

MmMapIoSpace function

MmMapIoSpaceEx function

MmMapLockedPages function

MmMapLockedPagesSpecifyCache function

MmMapLockedPagesWithReservedMapping function

MmMapMdl function

MmPageEntireDriver function

MmProbeAndLockPages function

MmProbeAndLockSelectedPages function

MmProtectMdlSystemAddress function

MmQuerySystemSize function

MmResetDriverPaging function

MmSizeOfMdl function

MmUnlockPagableImageSection function

MmUnlockPages function

MmUnmapIoSpace function

MmUnmapLockedPages function

MmUnmapReservedMapping function

NtCommitComplete function

NtCommitEnlistment function

NtCommitTransaction function

NtCreateEnlistment function

NtCreateResourceManager function

NtCreateTransaction function

NtCreateTransactionManager function

NtEnumerateTransactionObject function

NtGetNotificationResourceManager function

NtOpenEnlistment function

NtOpenResourceManager function

NtOpenTransaction function

NtOpenTransactionManager function

NtPowerInformation function

NtPrepareComplete function

NtPrepareEnlistment function

NtPrePrepareComplete function

NtPrePrepareEnlistment function

NtQueryInformationEnlistment function

NtQueryInformationResourceManager function

NtQueryInformationTransaction function

NtQueryInformationTransactionManager function

NtReadOnlyEnlistment function

NtRecoverEnlistment function

NtRecoverResourceManager function

NtRecoverTransactionManager function

NtRenameTransactionManager function

NtRollbackComplete function

NtRollbackEnlistment function

NtRollbackTransaction function

NtRollforwardTransactionManager function

NtSetInformationEnlistment function

NtSetInformationResourceManager function

NtSetInformationTransaction function

NtSetInformationTransactionManager function

NtSinglePhaseReject function

ObCloseHandle function

ObDereferenceObject macro

ObDereferenceObjectDeferDelete function

ObDereferenceObjectDeferDeleteWithTag function

ObDereferenceObjectWithTag macro

ObfReferenceObject function

ObGetObjectSecurity function

ObReferenceObject macro

ObReferenceObjectByHandle function

ObReferenceObjectByHandleWithTag function

ObReferenceObjectByPointer function

ObReferenceObjectByPointerWithTag function

ObReferenceObjectWithTag macro

ObRegisterCallbacks function

ObReleaseObjectSecurity function

ObUnRegisterCallbacks function

PALLOCATE_ADAPTER_CHANNEL callback function

PALLOCATE_ADAPTER_CHANNEL_EX callback function

PALLOCATE_COMMON_BUFFER callback function

PALLOCATE_COMMON_BUFFER_EX callback function

PBUILD_MDL_FROM_SCATTER_GATHER_LIST callback function

PBUILD_SCATTER_GATHER_LIST callback function

PBUILD_SCATTER_GATHER_LIST_EX callback function

PCALCULATE_SCATTER_GATHER_LIST_SIZE callback function

PCANCEL_ADAPTER_CHANNEL callback function

PCANCEL_MAPPED_TRANSFER callback function

PCI_MSIX_MASKUNMASK_ENTRY callback function

PCI_MSIX_SET_ENTRY callback function

PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK callback function

PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK callback function

PCLFS_CLIENT_LOG_UNPINNED_CALLBACK callback function

PCONFIGURE_ADAPTER_CHANNEL callback function

PDEVICE_RESET_HANDLER callback function

PFLUSH_ADAPTER_BUFFERS callback function

PFLUSH_ADAPTER_BUFFERS_EX callback function

PFREE_ADAPTER_CHANNEL callback function

PFREE_ADAPTER_OBJECT callback function

PFREE_COMMON_BUFFER callback function

PFREE_MAP_REGISTERS callback function

PGET_DMA_ADAPTER_INFO callback function

PGET_DMA_ALIGNMENT callback function

PGET_DMA_TRANSFER_INFO callback function

PGET_SCATTER_GATHER_LIST callback function

PGET_SCATTER_GATHER_LIST_EX callback function

PINITIALIZE_DMA_TRANSFER_CONTEXT callback function

PINTERFACE_DEREFERENCE callback function

PINTERFACE_REFERENCE callback function

PMAP_TRANSFER callback function

PMAP_TRANSFER_EX callback function

PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK callback function

PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK callback function

PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK callback function

PO_FX_COMPONENT_IDLE_STATE_CALLBACK callback function

PO_FX_COMPONENT_PERF_STATE_CALLBACK callback function

PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK callback function

PO_FX_DEVICE_POWER_REQUIRED_CALLBACK callback function

PO_FX_POWER_CONTROL_CALLBACK callback function

POB_POST_OPERATION_CALLBACK callback function

POB_PRE_OPERATION_CALLBACK callback function

PoCallDriver function

PoClearPowerRequest function

PoCreatePowerRequest function

PoDeletePowerRequest function

PoEndDeviceBusy function

PoFxActivateComponent function

PoFxCompleteDevicePowerNotRequired function

PoFxCompleteIdleCondition function

PoFxCompleteIdleState function

PoFxIdleComponent function

PoFxIssueComponentPerfStateChange function

PoFxIssueComponentPerfStateChangeMultiple function

PoFxNotifySurprisePowerOn function

PoFxPowerControl function

PoFxPowerOnCrashdumpDevice function

PoFxQueryCurrentComponentPerfState function

PoFxRegisterComponentPerfStates function

PoFxRegisterCrashdumpDevice function

PoFxRegisterDevice function

PoFxReportDevicePoweredOn function

PoFxSetComponentLatency function

PoFxSetComponentResidency function

PoFxSetComponentWake function

PoFxSetDeviceIdleTimeout function

PoFxSetTargetDripsDevicePowerState function

PoFxStartDevicePowerManagement function

PoFxUnregisterDevice function

PoGetSystemWake function

PopEntryList function

PoQueryWatchdogTime function

PoRegisterDeviceForIdleDetection function

PoRegisterPowerSettingCallback function

PoRegisterSystemState function

PoRequestPowerIrp function

PoSetDeviceBusyEx function

PoSetPowerRequest function

PoSetPowerState function

PoSetSystemState function

PoSetSystemWake function

PoStartDeviceBusy function

PoStartNextPowerIrp function

PoUnregisterPowerSettingCallback function

PoUnregisterSystemState function

PPUT_DMA_ADAPTER callback function

PPUT_SCATTER_GATHER_LIST callback function

PREAD_DMA_COUNTER callback function

PREENUMERATE_SELF callback function

ProbeForRead function

ProbeForWrite function

PROCESSOR_HALT_ROUTINE callback function

PsCreateSystemThread function

PsGetCurrentThread function

PsGetVersion function

PsQueryTotalCycleTimeProcess function

PsTerminateSystemThread function

PTM_RM_NOTIFICATION callback function

PushEntryList function

READ_PORT_BUFFER_UCHAR function

READ_PORT_BUFFER_ULONG function

READ_PORT_BUFFER_USHORT function

READ_PORT_UCHAR function

READ_PORT_ULONG function

READ_PORT_USHORT function

READ_REGISTER_BUFFER_UCHAR function

READ_REGISTER_BUFFER_ULONG function

READ_REGISTER_BUFFER_USHORT function

READ_REGISTER_UCHAR function

READ_REGISTER_ULONG function

READ_REGISTER_USHORT function

RemoveEntryList function

RemoveHeadList function

RemoveTailList function

REQUEST_POWER_COMPLETE callback function

RTL_QUERY_REGISTRY_ROUTINE callback function

RtlAnsiStringToUnicodeSize macro

RtlAnsiStringToUnicodeString function

RtlAppendUnicodeStringToString function

RtlAppendUnicodeToString function

RtlAreBitsClear function

RtlAreBitsSet function

RtlCheckBit macro

RtlCheckRegistryKey function

RtlClearAllBits function

RtlClearBit function

RtlClearBits function

RtlCmDecodeMemIoResource function

RtlCmEncodeMemIoResource function

RtlCompareMemory function

RtlCompareUnicodeString function

RtlConvertLongToLargeInteger function

RtlConvertUlongToLargeInteger function

RtlCopyMemory macro

RtlCopyUnicodeString function

RtlCreateRegistryKey function

RtlCreateSecurityDescriptor function

RtlDeleteRegistryValue function

RtlDowncaseUnicodeChar function

RtlEqualMemory macro

RtlEqualUnicodeString function

RtlFillMemory macro

RtlFindClearBits function

RtlFindClearBitsAndSet function

RtlFindClearRuns function

RtlFindFirstRunClear function

RtlFindLastBackwardRunClear function

RtlFindLeastSignificantBit function

RtlFindLongestRunClear function

RtlFindMostSignificantBit function

RtlFindNextForwardRunClear function

RtlFindSetBits function

RtlFindSetBitsAndClear function

RtlFreeAnsiString function

RtlFreeUnicodeString function

RtlGetVersion function

RtlGUIDFromString function

RtlHashUnicodeString function

RtlInitAnsiString function

RtlInitializeBitMap function

RtlInitString function

RtlInitStringEx function

RtlInitUnicodeString function

RtlInt64ToUnicodeString function

RtlIntegerToUnicodeString function

RtlIntPtrToUnicodeString macro

RtlIoDecodeMemIoResource function

RtlIoEncodeMemIoResource function

RtlIsNtDdiVersionAvailable function

RtlIsServicePackVersionInstalled function

RtlLengthSecurityDescriptor function

RtlMoveMemory macro

RtlNumberOfClearBits function

RtlNumberOfSetBits function

RtlNumberOfSetBitsUlongPtr function

RtlPrefetchMemoryNonTemporal function

RtlQueryRegistryValues function

RtlSecureZeroMemory function

RtlSetAllBits function

RtlSetBit function

RtlSetBits function

RtlSetDaclSecurityDescriptor function

RtlStringFromGUID function

RtlTestBit function

RtlTimeFieldsToTime function

RtlTimeToTimeFields function

RtlUlongByteSwap macro

RtlUlonglongByteSwap macro

RtlUnicodeStringToAnsiSize macro

RtlUnicodeStringToAnsiString function

RtlUnicodeStringToInteger function

RtlUnicodeToUTF8N function

RtlUpcaseUnicodeChar function

RtlUshortByteSwap macro

RtlUTF8ToUnicodeN function

RtlValidRelativeSecurityDescriptor function

RtlValidSecurityDescriptor function

RtlVerifyVersionInfo function

RtlWriteRegistryValue function

RtlxAnsiStringToUnicodeSize function

RtlxUnicodeStringToAnsiSize function

RtlZeroMemory macro

SeAccessCheck function

SeAssignSecurity function

SeAssignSecurityEx function

SeDeassignSecurity function

SET_D3COLD_SUPPORT callback function

SeValidSecurityDescriptor function

TmCommitComplete function

TmCommitEnlistment function

TmCommitTransaction function

TmCreateEnlistment function

TmDereferenceEnlistmentKey function

TmEnableCallbacks function

TmGetTransactionId function

TmInitializeTransactionManager function

TmIsTransactionActive function

TmPrepareComplete function

TmPrepareEnlistment function

TmPrePrepareComplete function

TmPrePrepareEnlistment function

TmReadOnlyEnlistment function

TmRecoverEnlistment function

TmRecoverResourceManager function

TmRecoverTransactionManager function

TmReferenceEnlistmentKey function

TmRenameTransactionManager function

TmRequestOutcomeEnlistment function

TmRollbackComplete function

TmRollbackEnlistment function

TmRollbackTransaction function

TmSinglePhaseReject function

TRANSLATE_BUS_ADDRESS callback function

VslCreateSecureSection function

VslDeleteSecureSection function

WmiQueryTraceInformation function

WmiTraceMessage function

WmiTraceMessageVa function
WRITE_PORT_BUFFER_UCHAR function
WRITE_PORT_BUFFER_ULONG function
WRITE_PORT_BUFFER_USHORT function
WRITE_PORT_UCHAR function
WRITE_PORT_ULONG function
WRITE_PORT_USHORT function
WRITE_REGISTER_BUFFER_UCHAR function
WRITE_REGISTER_BUFFER_ULONG function
WRITE_REGISTER_BUFFER_ULONG64 function
WRITE_REGISTER_BUFFER_USHORT function
WRITE_REGISTER_UCHAR function
WRITE_REGISTER_ULONG function
WRITE_REGISTER_USHORT function
ZwClose function
ZwCommitComplete function
ZwCommitEnlistment function
ZwCommitTransaction function
ZwCreateDirectoryObject function
ZwCreateEnlistment function
ZwCreateFile function
ZwCreateKey function
ZwCreateKeyTransacted function
ZwCreateResourceManager function
ZwCreateSection function
ZwCreateTransaction function
ZwCreateTransactionManager function
ZwDeleteKey function
ZwDeleteValueKey function
ZwEnumerateKey function
ZwEnumerateTransactionObject function
ZwEnumerateValueKey function

ZwFlushKey function

ZwGetNotificationResourceManager function

ZwLoadDriver function

ZwMakeTemporaryObject function

ZwMapViewOfSection function

ZwOpenEnlistment function

ZwOpenEvent function

ZwOpenFile function

ZwOpenKey function

ZwOpenKeyEx function

ZwOpenKeyTransacted function

ZwOpenKeyTransactedEx function

ZwOpenResourceManager function

ZwOpenSection function

ZwOpenSymbolicLinkObject function

ZwOpenTransaction function

ZwOpenTransactionManager function

ZwPrepareComplete function

ZwPrepareEnlistment function

ZwPrePrepareComplete function

ZwPrePrepareEnlistment function

ZwQueryFullAttributesFile function

ZwQueryInformationEnlistment function

ZwQueryInformationFile function

ZwQueryInformationResourceManager function

ZwQueryInformationTransaction function

ZwQueryInformationTransactionManager function

ZwQueryKey function

ZwQuerySymbolicLinkObject function

ZwQueryValueKey function

ZwReadFile function

ZwReadOnlyEnlistment function

- ZwRecoverEnlistment function
- ZwRecoverResourceManager function
- ZwRecoverTransactionManager function
- ZwRollbackComplete function
- ZwRollbackEnlistment function
- ZwRollbackTransaction function
- ZwRollforwardTransactionManager function
- ZwSetInformationEnlistment function
- ZwSetInformationFile function
- ZwSetInformationResourceManager function
- ZwSetInformationTransaction function
- ZwSetValueKey function
- ZwSinglePhaseReject function
- ZwUnloadDriver function
- ZwUnmapViewOfSection function
- ZwWriteFile function

Wdmsec.h

- WdmlibIoCreateDeviceSecure function
- WdmlibIoValidateDeviceIoControlAccess function
- WdmlibRtlInitUnicodeStringEx function

Wmidata.h

Wmilib.h

- WMIGUIDREGINFO structure
- WMILIB_CONTEXT structure
- WMI_EXECUTE_METHOD_CALLBACK callback function
- WMI_FUNCTION_CONTROL_CALLBACK callback function
- WMI_QUERY_DATABLOCK_CALLBACK callback function
- WMI_QUERY_REGINFO_CALLBACK callback function
- WMI_SET_DATABLOCK_CALLBACK callback function
- WMI_SET_DATAITEM_CALLBACK callback function
- WmiCompleteRequest function
- WmiFireEvent function

- WmiSystemControl function

Wmistr.h

- WMIREGGUIDW structure
- WMIREGINFOW structure
- WNODE_HEADER structure
- WNODE_ALL_DATA structure
- WNODE_EVENT_ITEM structure
- WNODE_EVENT_REFERENCE structure
- WNODE_METHOD_ITEM structure
- WNODE_SINGLE_INSTANCE structure
- WNODE_SINGLE_ITEM structure
- WNODE_TOO_SMALL structure

Wudfwdm.h

- CM_FULL_RESOURCE_DESCRIPTOR structure
- CM_PARTIAL_RESOURCE_DESCRIPTOR structure
- CM_PARTIAL_RESOURCE_LIST structure
- CM_RESOURCE_LIST structure
- INTERFACE_TYPE enumeration
- KINTERRUPT_MODE enumeration
- MEMORY_CACHING_TYPE enumeration
- PINTERFACE_DEREFERENCE callback function
- PINTERFACE_REFERENCE callback function
- DEVICE_REGISTRY_PROPERTY enumeration
- IO_STATUS_BLOCK structure
- OBJECT_ATTRIBUTES structure
- PO_FX_COMPONENT_IDLE_STATE structure
- PO_FX_COMPONENT_PERF_INFO structure
- PO_FX_COMPONENT_PERF_SET structure
- PO_FX_COMPONENT_V1 structure
- PO_FX_COMPONENT_V2 structure
- PO_FX_DEVICE_V1 structure
- PO_FX_DEVICE_V2 structure

PO_FX_PERF_STATE structure

PO_FX_PERF_STATE_CHANGE structure

PO_FX_PERF_STATE_TYPE enumeration

PO_FX_PERF_STATE_UNIT enumeration

POOL_TYPE enumeration

UNICODE_STRING structure

AppendTailList function

ARGUMENT_PRESENT macro

DRIVER_INITIALIZE callback function

InitializeListHead function

InitializeObjectAttributes macro

InsertHeadList function

InsertTailList function

IsListEmpty function

PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK callback function

PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK callback function

PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK callback function

PO_FX_COMPONENT_IDLE_STATE_CALLBACK callback function

PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK callback function

PO_FX_DEVICE_POWER_REQUIRED_CALLBACK callback function

PO_FX_POWER_CONTROL_CALLBACK callback function

PopEntryList function

PushEntryList function

READ_REGISTER_BUFFER_UCHAR function

READ_REGISTER_BUFFER_ULONG function

READ_REGISTER_BUFFER_USHORT function

READ_REGISTER_UCHAR function

READ_REGISTER_ULONG function

READ_REGISTER_USHORT function

RemoveEntryList function

RemoveHeadList function

RemoveTailList function

RtlAppendUnicodeStringToString function

RtlAppendUnicodeToString function

RtlCompareUnicodeString function

RtlCopyUnicodeString function

RtlEqualUnicodeString function

RtlFreeUnicodeString function

RtlInitUnicodeString function

RtlInt64ToUnicodeString function

RtlIntegerToUnicodeString function

RtlIntPtrToUnicodeString macro

RtlUnicodeStringToInteger function

WRITE_REGISTER_BUFFER_UCHAR function

WRITE_REGISTER_BUFFER_ULONG function

WRITE_REGISTER_BUFFER_USHORT function

WRITE_REGISTER_UCHAR function

WRITE_REGISTER_ULONG function

WRITE_REGISTER_USHORT function

Windows kernel

1/16/2019 • 291 minutes to read • Edit Online

Standard driver routines must be implemented by your driver. Driver support routines are routines that the Windows operating system provides. Drivers do not use Microsoft Win32 routines; instead, they use the driver support routines that this section describes. The driver support routines in this section are organized by kernel-mode managers and libraries.

- Standard Driver Routines Here are the required and optional routines that you must implement in your driver to respond to calls from Windows or other drivers. When these routines are called, your code must respond to the call and return the appropriate data in a timely manner.

  The following routines are required for all drivers:

  - DriverEntry
  - AddDevice
  - DispatchXxx (routines beginning with "Dispatch" such as DispatchCreate)
  - DriverUnload

    The other routines are optional, but you may need to implement them depending on your driver type and the location of your driver in the device stack.

    For more information about standard driver routines, see Introduction to Standard Driver Routines.

- Driver Support Routines

  Driver support routines are routines that the Windows operating system provides for kernel-mode drivers to use. Drivers do not use Microsoft Win32 routines; instead, they use the driver support routines that this section describes.

  The driver support routines in this section are organized by kernel-mode managers and libraries.

  The following kernel-mode managers provide support for drivers:

  - Object Manager Routines
  - Memory Manager Routines
  - Process and Thread Manager Routines
  - I/O Manager Routines
  - Power Manager Routines
  - Configuration Manager Routines
  - Kernel Transaction Manager Routines
  - Security Reference Monitor Routines

    The following kernel-mode libraries provide support for drivers:

  - Core Kernel Library Support Routines
  - Executive Library Support Routines
  - Run-Time Library (RTL) Routines
  - Safe String Library Routines
  - Safe Integer Library Routines
  - Direct Memory Access (DMA) Library Routines

- Hardware Abstraction Layer (HAL) Library Routines
- Common Log File System (CLFS) Routines
- Windows Management Instrumentation (WMI) Library Support Routines
- ZwXxx Routines
- Auxiliary Kernel-Mode Library Routines and Structures
- Processor Group Compatibility Library

Public Headers for Windows Kernel

To develop Windows kernel, you need these headers:

- aux_klib.h
- buffring.h
- evntrace.h
- hwnclx.h
- ioaccess.h
- iointex.h
- ntddk.h
- ntddsfio.h
- ntddsysenv.h
- ntimage.h
- ntintsafe.h
- ntnls.h
- ntpoapi.h
- ntstrsafe.h
- pcivirt.h
- pep_x.h
- pepevents.h
- pepfx.h
- poclass.h
- procgrp.h
- pwmutil.h
- vpci.h
- wdm.h
- wdmsec.h
- wmidata.h
- wmilib.h
- wmistr.h

For the programming guide, see Windows kernel.

Initialization and Unload

This section summarizes kernel-mode support routines that can be called by drivers from their DriverEntry, AddDevice, Reinitialize, or Unload routines.

ROUTINES FOR ...	ROUTINE

ROUTINES FOR ...	ROUTINE
Obtaining and reporting hardware configuration information about a driver's devices and the current platform.	IoGetDeviceProperty IoReportDetectedDevice IoReportResourceForDetection IoGetDmaAdapter IoGetConfigurationInformation HalExamineMBR IoReadPartitionTable IoInvalidateDeviceRelations IoInvalidateDeviceState IoRegisterPlugPlayNotification IoUnregisterPlugPlayNotification IoRequestDeviceEject IoReportTargetDeviceChange
Obtaining and reporting configuration information, and for registering interfaces in the registry.	IoGetDeviceProperty IoOpenDeviceInterfaceRegistryKey IoOpenDeviceRegistryKey IoRegisterDeviceInterface IoSetDeviceInterfaceState RtlCheckRegistryKey RtlCreateRegistryKey RtlQueryRegistryValues RtlWriteRegistryValue RtlDeleteRegistryValue InitializeObjectAttributes ZwCreateKey ZwOpenKey ZwQueryKey ZwEnumerateKey ZwEnumerateValueKey ZwQueryValueKey ZwSetValueKey ZwFlushKey ZwDeleteKey ZwClose
Setting up and freeing the objects and resources that drivers might use.	IoCreateDevice IoDeleteDevice IoGetDeviceObjectPointer

ROUTINES FOR ...	ROUTINE
	IoAttachDeviceToDeviceStack
	IoGetAttachedDeviceReference
	IoDetachDevice
	IoAllocateDriverObjectExtension
	IoGetDriverObjectExtension
	IoRegisterDeviceInterface
	IoIsWdmVersionAvailable
	IoDeleteSymbolicLink
	IoAssignArcName
	IoDeassignArcName
	IoSetShareAccess
	IoConnectInterrupt
	IoDisconnectInterrupt
	IoConnectInterruptEx
	IoDisconnectInterruptEx
	IoInitializeDpcRequest
	IoReadPartitionTable
	IoSetPartitionInformation
	IoWritePartitionTable
	IoCreateController
	IoDeleteController
	KeInitializeSpinLock
	KeInitializeDpc
	KeInitializeTimer
	KeInitializeTimerEx
	KeInitializeEvent
	ExInitializeFastMutex
	KeInitializeMutex
	KeInitializeSemaphore
	IoCreateNotificationEvent
	IoCreateSynchronizationEvent
	PsCreateSystemThread
	PsTerminateSystemThread
	KeSetBasePriorityThread
	KeSetPriorityThread
	MmIsThisAnNtAsSystem
	MmQuerySystemSize
	ExInitializeNPagedLookasideList
	ExInitializePagedLookasideList
	ExInitializeResourceLite
	ExReinitializeResourceLite

ROUTINES FOR ...	ROUTINE
	ExDeleteResourceLite ObReferenceObjectByHandle ObReferenceObjectByPointer ObReferenceObject ObDereferenceObject RtlInitString RtlInitAnsiString RtlInitUnicodeString InitializeObjectAttributes ZwCreateDirectoryObject ZwCreateFile ZwCreateKey ZwDeleteKey ZwMakeTemporaryObject ZwClose PsGetVersion ObGetObjectSecurity ObReleaseObjectSecurity
Initializing driver-managed internal queues.	KeInitializeSpinLock InitializeListHead ExInitializeSListHead KeInitializeDeviceQueue IoCsqInitialize

The following routines are reserved for system use. Do not use them in your driver.

- HalAcquireDisplayOwnership
- HalAllocateAdapterChannel. Use AllocateAdapterChannel instead.
- HalAllocateCrashDumpRegisters
- HalAllocateMapRegisters
- HalGetScatterGatherList. Use GetScatterGatherList instead.
- HalMakeBeep
- HalPutDmaAdapter. Use PutDmaAdapter instead.
- HalPutScatterGatherList. Use PutScatterGatherList instead.

The following obsolete routines are exported only to support existing drivers:

OBSOLETE ROUTINE	REPLACEMENT
HalAllocateCommonBuffer	See AllocateCommonBuffer instead.

OBSOLETE ROUTINE	REPLACEMENT
HalAssignSlotResources	Drivers of PnP devices are assigned resources by the PnP manager, which passes resource lists with each IRP_MN_START_DEVICE request. Drivers that must support a legacy device that cannot be enumerated by the PnP manager should use IoReportDetectedDevice and IoReportResourceForDetection.
HalFreeCommonBuffer	See FreeCommonBuffer instead.
HalGetAdapter	See IoGetDmaAdapter instead.
HalGetBusData	Instead, use IRP_MN_QUERY_INTERFACE to query the GUID_BUS_INTERFACE_STANDARD interface. This query request returns a function pointer to GetBusData, which can be used to read from the configuration space of a given device.
HalGetBusDataByOffset	Instead, use IRP_MN_QUERY_INTERFACE to query the GUID_BUS_INTERFACE_STANDARD interface. This query request returns a function pointer to GetBusData, which can be used to read from the configuration space of a given device.
HalGetDmaAlignmentRequirement	See GetDmaAlignment instead.
HalGetInterruptVector	Drivers of PnP devices are assigned resources by the PnP manager, which passes resource lists with each IRP_MN_START_DEVICE request. Drivers that must support a legacy device that cannot be enumerated by the PnP manager should use IoReportDetectedDevice and IoReportResourceForDetection.
HalReadDmaCounter	See ReadDmaCounter instead.
HalSetBusData	Instead, use IRP_MN_QUERY_INTERFACE to query the GUID_BUS_INTERFACE_STANDARD interface. This query request returns a function pointer to SetBusData, which can be used to write to the configuration space of a given device.
HalSetBusDataByOffset	Instead, use IRP_MN_QUERY_INTERFACE to query the GUID_BUS_INTERFACE_STANDARD interface. This query request returns a function pointer to SetBusData, which can be used to write to the configuration space of a given device.
HalTranslateBusAddress	The PnP manager passes lists of raw and translated resources in its IIRP_MN_START_DEVICE request for each device. Consequently, in most cases there is no need to translate bus addresses. However, if translation is required, use IRP_MN_QUERY_INTERFACE to query the GUID_BUS_INTERFACE_STANDARD interface. The query request returns a function pointer to TranslateBusAddress, which can be used to translates addresses on the parent bus to logical addresses.

These obsolete routines are included in the Ntddk.h header file.

The following routine is not supported and should not be used:

- HalReturnToFirmware

Windows performs resource balancing of PCI bus resources to open an address region for a plugged device. The rebalance operation causes the bus data for some driver's device to move dynamically (between IRP_MN_STOP_DEVICE and IRP_MN_START_DEVICE calls). Therefore, a driver must not access the bus data directly. Instead the driver must pass it down to the lower bus driver because it knows the location of the device.

For more information, see Stopping a Device to Rebalance Resources.

Registry

Used for obtaining and reporting configuration information, and for registering interfaces in the registry.

FUNCTION	DESCRIPTION
IoGetDeviceProperty	Retrieves device setup information from the registry. Use this routine, rather than accessing the registry directly, to insulate a driver from differences across platforms and from possible changes in the registry structure.
IoOpenDeviceInterfaceRegistryKey	Returns a handle to a registry key for storing information about a particular device interface.
IoOpenDeviceRegistryKey	Returns a handle to a device-specific or a driver-specific registry key for a particular device instance.
IoRegisterDeviceInterface	Registers device functionality (a device interface) that a driver will enable for use by applications or other system components. The I/O manager creates a registry key for the device interface. Drivers can access persistent storage under this key using IoOpenDeviceInterfaceRegistryKey.
IoSetDeviceInterfaceState	Enables or disables a previously registered device interface. Applications and other system components can open only interfaces that are enabled.
RtlCheckRegistryKey	Returns STATUS_SUCCESS if a key exists in the registry along the given relative path.
RtlCreateRegistryKey	Adds a key object in the registry along the given relative path.
RtlQueryRegistryValues	Gives a driver-supplied callback read-only access to the entries for a specified value name, along a specified relative path, in the registry, after the callback routine is given control.
RtlWriteRegistryValue	Writes caller-supplied data into the registry along the specified relative path at the given value name.
RtlDeleteRegistryValue	Removes the specified value name (and the associated value entries) from the registry along the given relative path.
InitializeObjectAttributes	Sets up a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a ZwCreateXxx or ZwOpenXxx routine.

FUNCTION	DESCRIPTION
ZwCreateKey	Creates a new key in the registry with the given object's attributes, allowed access, and creation options (such as whether the key is created again when the system is booted). Alternatively, opens an existing key and returns a handle for the key object.
ZwOpenKey	Returns a handle for a key in the registry given the object's attributes (which must include a name for the key) and the desired access to the object.
ZwQueryKey	Returns information about the class of a key, and the number and sizes of its subkeys. This information includes, for example, the length of subkey names and the size of value entries.
ZwEnumerateKey	Returns the specified information about the subkey, as selected by a zero-based index, of an opened key in the registry.
ZwEnumerateValueKey	Returns the specified information about the value entry of a subkey, as selected by a zero-based index, of an opened key in the registry.
ZwQueryValueKey	Returns the value entry for an opened key in the registry.
ZwSetValueKey	Replaces (or creates) a value entry for an opened key in the registry.
ZwFlushKey	Forces changes made by ZwCreateKey or ZwSetValueKey for the opened key object to be written to disk.
ZwDeleteKey	Removes a key and its value entries from the registry as soon as the key is closed.
ZwClose	Releases the handle for an opened object, causing the handle to become invalid and decrementing the reference count of the object handle.

Objects and Resources

Used for setting up and freeing the objects and resources that drivers might use.

FUNCTION	DESCRIPTION
IoCreateDevice	Initializes a device object, which represents a physical, virtual, or logical device for which the driver is being loaded into the system. Then it allocates space for the driver-defined device extension associated with the device object.
IoDeleteDevice	Removes a device object from the system when the underlying device is removed from the system.
IoGetDeviceObjectPointer	Requests access to a named device object and returns a pointer to that device object if the requested access is granted. Also returns a pointer to the file object referenced by the named device object. In effect, this routine establishes a connection between the caller and the next-lower-level driver.

FUNCTION	DESCRIPTION
IoAttachDeviceToDeviceStack	Attaches the caller's device object to the highest device object in a chain of drivers and returns a pointer to the previously highest device object. I/O requests bound for the target device are routed first to the caller.
IoGetAttachedDeviceReference	Returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.
IoDetachDevice	Releases an attachment between the caller's device object and a target driver's device object.
IoAllocateDriverObjectExtension	Allocates a per-driver context area with a given unique identifier.
IoGetDriverObjectExtension	Retrieves a previously allocated per-driver context area.
IoRegisterDeviceInterface	Registers device functionality (a device interface) that a driver will enable for use by applications or other system components. The I/O manager creates a registry key for the device interface. Drivers can access persistent storage under this key by calling IoOpenDeviceInterfaceRegistryKey.
IoIsWdmVersionAvailable	Checks whether a given WDM version is supported by the operating system.
IoDeleteSymbolicLink	Releases a symbolic link between a device object name and a user-visible name.
IoAssignArcName	Sets up a symbolic link between a named device object (such as a tape, floppy, or CD-ROM) and the corresponding ARC name for the device.
IoDeassignArcName	Releases the symbolic link created by calling IoAssignArcName.
IoSetShareAccess	Sets the access allowed to a given file object that represents a device. (Only highest-level drivers can call this routine.)
IoConnectInterrupt	Registers a driver's interrupt-handling routine. Drivers should use IoConnectInterruptEx instead.
IoDisconnectInterrupt	Unregisters an interrupt-handling routine that IoConnectInterrupt registered.
IoConnectInterruptEx	Registers a driver's interrupt-handling routine. Drivers can register either an InterruptService routine for line-based interrupts or an InterruptMessageService routine for message-signaled interrupts.

FUNCTION	DESCRIPTION
IoDisconnectInterruptEx	Unregisters an interrupt-handling routine that IoConnectInterruptEx registered.
IoInitializeDpcRequest	Associates a driver-supplied DpcForIsr routine with a given device object, so that the DpcForIsr routine can complete interrupt-driven I/O operations.
IoReadPartitionTable	Returns a list of partitions on a disk with a given sector size.
IoSetPartitionInformation	Sets the partition type and number for a (disk) partition.
IoWritePartitionTable	Writes partition tables for a disk, given the device object that represents the disk, the sector size, and a pointer to a buffer containing the drive layout structure.
IoCreateController	Initializes a controller object representing a physical device controller that is shared by two or more similar devices that have the same driver, and specifies the size of the controller extension.
IoDeleteController	Removes a controller object from the system.
KeInitializeSpinLock	Initializes a variable of type KSPIN_LOCK.
KeInitializeDpc	Initializes a DPC object, setting up a driver-supplied CustomDpc routine that can be called with a given context.
KeInitializeTimer	Initializes a notification timer object to the Not-Signaled state.
KeInitializeTimerEx	Initializes a notification or synchronization timer object to the Not-Signaled state.
KeInitializeEvent	Initializes an event object as a synchronization (single waiter) or notification (multiple waiters) type event and sets up its initial state (Signaled or Not-Signaled).
ExInitializeFastMutex	Initializes a fast mutex variable that is used to synchronize mutually exclusive access to a shared resource by a set of threads.
KeInitializeMutex	Initializes a mutex object as set to the Signaled state.
KeInitializeSemaphore	Initializes a semaphore object to a given count and specifies an upper bound for the count.
IoCreateNotificationEvent	Initializes a named notification event to be used to synchronize access between two or more components. Notification events are not automatically reset.
IoCreateSynchronizationEvent	Initializes a named synchronization event to be used to serialize access to hardware between two otherwise unrelated drivers.

FUNCTION	DESCRIPTION
PsCreateSystemThread	Creates a kernel-mode thread that is associated with a given process object or with the default system process. Returns a handle for the thread.
PsTerminateSystemThread	Terminates the current thread and satisfies as many waits as possible for the current thread object.
KeSetBasePriorityThread	Sets up the run-time priority, relative to the system process, for a driver-created thread.
KeSetPriorityThread	Sets up the run-time priority for a driver-created thread with a real-time priority attribute.
MmIsThisAnNtAsSystem	Returns TRUE if the current platform is a server, indicating that more resources are likely to be necessary to process I/O requests than if the machine were a client.
MmQuerySystemSize	Returns an estimate (small, medium, or large) of the amount of memory available on the current platform.
ExInitializeNPagedLookasideList	Initializes a lookaside list of nonpaged memory. After a successful initialization, fixed-size blocks can be allocated from and freed to the lookaside list.
ExInitializePagedLookasideList	Initializes a lookaside list of paged memory. After a successful initialization, fixed-size blocks can be allocated from and freed to the lookaside list.
ExInitializeResourceLite	Initializes a resource, for which the caller provides the storage, to be used for synchronization by a set of threads.
ExReinitializeResourceLite	Reinitializes an existing resource variable.
ExDeleteResourceLite	Deletes a caller-initialized resource from the system's resource list.
ObReferenceObjectByHandle	Returns a pointer to the object body and handle information (attributes and granted access rights), given the handle for an object, the object's type, and a mask. Specifies the desired access to the object and the preferred access mode. A successful call increments the reference count for the object.
ObReferenceObjectByPointer	Increments the reference count for an object so the caller can ensure that the object is not removed from the system while the caller is using it.
ObReferenceObject	Increments the reference count for an object, given a pointer to the object.
ObDereferenceObject	Releases a reference to an object (decrements the reference count), given a pointer to the object body.
RtlInitString	Initializes a counted string in a buffer.

FUNCTION	DESCRIPTION
RtlInitAnsiString	Initializes a counted ANSI string in a buffer.
RtlInitUnicodeString	Initializes a counted Unicode string in a buffer.
InitializeObjectAttributes	Initializes a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a ZwCreateXxx or ZwOpenXxx routine.
ZwCreateDirectoryObject	Creates or opens a directory object with a specified set of object attributes and requests one or more types of access for the caller. Returns a handle for the directory object.
ZwCreateFile	Creates or opens a file object that represents a physical, logical, or virtual device, a directory, a data file, or a volume. Returns a handle for the file object.
ZwCreateKey	Creates or opens a key object in the registry and returns a handle for the key object.
ZwDeleteKey	Deletes an existing, open key in the registry after the last handle for the key is closed.
ZwMakeTemporaryObject	Resets the "permanent" attribute of an opened object, so that the object and its name can be deleted when the reference count for the object becomes zero.
ZwClose	Releases the handle for an opened object, causing the handle to become invalid, and decrements the reference count of the object handle.
PsGetVersion	Indicates whether the driver is running on a free or checked build, and optionally supplies information about the operating system version and build number.
ObGetObjectSecurity	Returns a buffered security descriptor for a given object.
ObReleaseObjectSecurity	Releases the security descriptor returned by ObGetObjectSecurity.

Initializing Driver-Managed Queues

Used for initializing driver-managed internal queues.

FUNCTION	DESCRIPTION
KeInitializeSpinLock	Initializes a variable of type KSPIN_LOCK. An initialized spin lock is a required parameter to the ExXxxInterlockedList routines.
InitializeListHead	Sets up a queue header for a driver's internal queue, given a pointer to driver-supplied storage for the queue header and queue.
ExInitializeSListHead	Sets up the queue header for a sequenced, interlocked, singly linked list.

FUNCTION	DESCRIPTION
KeInitializeDeviceQueue	Initializes a device queue object to a Not Busy state, setting up an associated spin lock for multiprocessor-safe access to device queue entries.
IoCsqInitialize	Initializes the dispatch table for a driver's cancel-safe IRP queue.

I/O Manager Routines

All kernel-mode drivers except video and SCSI miniport drivers and NDIS drivers call IoXxx routines.

References for the IoXxx routines are in alphabetical order.

For an overview of the functionality of these routines, see Summary of Kernel-Mode Support Routines.

The following routine is for system use. Do not use it in your driver.

IoUpdateDiskGeometry

Power Management Routines

The Windows power management architecture provides a comprehensive approach to power management supported at the component (subdevice) level, in addition to the system level and the device level.

Kernel-mode drivers call the PoXxx routines to perform power management for the devices that they control. This section contains the reference pages for these routines. The PoXxx routines are declared in the Wdm.h header file.

For more information about power management, see Power Management for Windows Drivers.

FUNCTION	DESCRIPTION
PoCallDriver	The PoCallDriver routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.)
PoClearPowerRequest	The PoClearPowerRequest routine decrements the count for the specified power request type.
PoCreatePowerRequest	The PoCreatePowerRequest routine creates a power request object.
PoDeletePowerRequest	The PoDeletePowerRequest routine deletes a power request object.
PoEndDeviceBusy	The PoEndDeviceBusy routine marks the end of a period of time in which the device is busy.
PoGetSystemWake	The PoGetSystemWake routine determines whether a specified IRP has been marked as waking the system from a sleeping state.
PoQueryWatchdogTime	The PoQueryWatchdogTime routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.

FUNCTION	DESCRIPTION
PoRegisterDeviceForIdleDetection	The PoRegisterDeviceForIdleDetection routine enables or cancels idle detection and sets idle time-out values for a device.
PoRegisterPowerSettingCallback	The PoRegisterPowerSettingCallback routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.
PoRegisterSystemState	The PoRegisterSystemState routine registers the system as busy due to certain activity.
PoRequestPowerIrp	The PoRequestPowerIrp routine allocates a power IRP and sends it to the top driver in the device stack for the specified device.
PoSetDeviceBusyEx	The PoSetDeviceBusyEx routine notifies the power manager that the device associated with the specified idle counter is busy.
PoSetPowerRequest	The PoSetPowerRequest routine increments the count for the specified power request type.
PoSetPowerState	The PoSetPowerState routine notifies the system of a change in the device power state for a device.
PoSetSystemState	Drivers call the PoSetSystemState routine to indicate that the system is active.
PoSetSystemWake	The PoSetSystemWake routine marks the specified IRP as one that contributed to waking the system from a sleep state.
PoStartDeviceBusy	The PoStartDeviceBusy routine marks the start of a period of time in which the device is busy.
PoStartNextPowerIrp	The PoStartNextPowerIrp routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)
PoUnregisterPowerSettingCallback	The PoUnregisterPowerSettingCallback routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.
PoUnregisterSystemState	The PoUnregisterSystemState routine cancels a system state registration created by PoRegisterSystemState.

Device Power Management

Starting with Windows 8, drivers can divide their device hardware into multiple logical components to enable fine-grained power management. A component has a set of power states that can be managed independently of the power states of other components in the same device. In the F0 state, the component is fully turned on. The component might support additional, low-power states F1, F2, and so on.

The power policy owner for a device is typically the device's function driver. To enable component-level power management, this driver registers the device with the power management framework (PoFx). By registering the

device, the driver assumes the responsibility for informing PoFx when a component is actively being used and when the component is idle. PoFx makes intelligent idle state choices for the device based on information about the component activity, latency tolerance, expected idle durations, and wake requirements. By controlling power usage at the component level, PoFx can reduce power requirements while preserving system responsiveness. For more information, see Component-Level Power Management.

Starting with Windows 8, drivers can divide their device hardware into multiple logical components to enable fine-grained power management. A component has a set of power states that can be managed independently of the power states of other components in the same device. In the F0 state, the component is fully turned on. The component might support additional, low-power states F1, F2, and so on.

The power policy owner for a device is typically the device's function driver. To enable component-level power management, this driver registers the device with the power management framework (PoFx). By registering the device, the driver assumes the responsibility for informing PoFx when a component is actively being used and when the component is idle. PoFx makes intelligent idle state choices for the device based on information about the component activity, latency tolerance, expected idle durations, and wake requirements. By controlling power usage at the component level, PoFx can reduce power requirements while preserving system responsiveness. For more information, see Component-Level Power Management.

This section describes the routines that are implemented by the power management framework (PoFx) to enable device power management. These routines are called by the driver that is the power policy owner (PPO) for a device. Typically, the function driver for a device is the PPO for this device.

FUNCTION	DESCRIPTION
PoFxActivateComponent	The PoFxActivateComponent routine increments the activation reference count on the specified component.
PoFxCompleteDevicePowerNotRequired	The PoFxCompleteDevicePowerNotRequired routine notifies the power management framework (PoFx) that the calling driver has completed its response to a call to the driver's DevicePowerNotRequiredCallback callback routine.
PoFxCompleteIdleCondition	The PoFxCompleteIdleCondition routine informs the power management framework (PoFx) that the specified component has completed a pending change to the idle condition.
PoFxCompleteIdleState	The PoFxCompleteIdleState routine informs the power management framework (PoFx) that the specified component has completed a pending change to an Fx state.
PoFxIdleComponent	The PoFxIdleComponent routine decrements the activation reference count on the specified component.
PoFxIssueComponentPerfStateChange	The PoFxIssueComponentPerfStateChange routine submits a request to place a device component in a particular performance state.
PoFxIssueComponentPerfStateChangeMultiple	The PoFxIssueComponentPerfStateChangeMultiple routine submits a request to change the performance states in multiple performance state sets simultaneously for a device component.
PoFxNotifySurprisePowerOn	The PoFxNotifySurprisePowerOn routine notifies the power management framework (PoFx) that a device was turned on as a side effect of supplying power to some other device.

FUNCTION	DESCRIPTION
PoFxPowerControl	The PoFxPowerControl routine sends a power control request to the power management framework (PoFx).
PoFxQueryCurrentComponentPerfState	The PoFxQueryCurrentComponentPerfState routine retrieves the active performance state in a component's performance state set.
PoFxRegisterComponentPerfStates	The PoFxRegisterComponentPerfStates routine registers a device component for performance state management by the power management framework (PoFx).
PoFxRegisterDevice	The PoFxRegisterDevice routine registers a device with the power management framework (PoFx).
PoFxReportDevicePoweredOn	The PoFxReportDevicePoweredOn routine notifies the power management framework (PoFx) that the device completed the requested transition to the D0 (fully on) power state.
PoFxSetComponentLatency	The PoFxSetComponentLatency routine specifies the maximum latency that can be tolerated in the transition from the idle condition to the active condition in the specified component.
PoFxSetComponentResidency	The PoFxSetComponentResidency routine sets the estimated time for how long a component is likely to remain idle after the component enters the idle condition.
PoFxSetComponentWake	The PoFxSetComponentWake routine indicates whether the driver arms the specified component to wake whenever the component enters the idle condition.
PoFxSetDeviceIdleTimeout	The PoFxSetDeviceIdleTimeout routine specifies the minimum time interval from when the last component of the device enters the idle condition to when the power management framework (PoFx) calls the driver's DevicePowerNotRequiredCallback routine.
PoFxSetTargetDripsDevicePowerState	This routine is called to notify the power manager of the device's target device power state for DRIPS. The driver can override the DRIPS constraint provided by the PEP.
PoFxStartDevicePowerManagement	The PoFxStartDevicePowerManagement routine completes the registration of a device with the power management framework (PoFx) and starts device power management.
PoFxUnregisterDevice	The PoFxUnregisterDevice routine removes the registration of a device from the power management framework (PoFx).

Device Power Management Callbacks

Device power management callbacks are the callback routines that are required by the power management framework (PoFx) to enable device power management. The driver that is the power policy owner for the device implements these callback routines. PoFx calls these routines to query and configure the power states of the components in the device.

CALLBACK	DESCRIPTION
ComponentActiveConditionCallback	The ComponentActiveConditionCallback callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.
ComponentIdleConditionCallback	The ComponentIdleConditionCallback callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
ComponentIdleStateCallback	The ComponentIdleStateCallback callback routine notifies the driver of a pending change to the Fx power state of the specified component.
ComponentPerfStateCallback	The ComponentPerfStateCallback callback routine notifies the driver that its request to change the performance state of a component is complete.
DevicePowerNotRequiredCallback	The DevicePowerNotRequiredCallback callback routine notifies the device driver that the device is not required to stay in the D0 power state.
DevicePowerRequiredCallback	The DevicePowerRequiredCallback callback routine notifies the device driver that the device must enter and remain in the D0 power state.
PowerControlCallback	The PowerControlCallback callback routine performs a power control operation that is requested by the power management framework (PoFx).

Platform extension plug-in (PEP) reference

Platform extension plug-ins (PEPs) provide interfaces for platform power management including device power management (DPM), processor power management (PPM), and, starting with Windows 10, ACPI runtime methods.

The types of notifications sent to platform extension plug-ins (PEPs) include: ACPI notifications

Device power management (DPM) notifications

Processor power management (PPM) notifications

PPM power control codes

INITIALIZATION FUNCTION	DESCRIPTION
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

INITIALIZATION FUNCTION	DESCRIPTION
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE	The PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE	The PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

PEP callback routines The callback routines are implemented by platform extension plug-ins, and are called by the Windows power management framework (PoFx).

CALLBACK FUNCTION	DESCRIPTION
AcceptAcpiNotification	An AcceptAcpiNotification event callback routine handles ACPI notifications from the Windows power management framework (PoFx).
AcceptDeviceNotification	An AcceptDeviceNotification event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).
AcceptProcessorNotification	An AcceptProcessorNotification event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).
PO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK	An EnumerateInterruptSource callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.
PROCESSOR_HALT_ROUTINE	A Halt callback routine transitions the processor to an idle state.
PowerOnDumpDeviceCallback	The PowerOnDumpDeviceCallback callback routine turns on the crash-dump device.

Power management framework (PoFx) routines

Some miscellaneous PoFx routines and structures will most likely end up in the power-management section of the Kernel-Mode Driver Architecture doc set.

INITIALIZATION FUNCTION	DESCRIPTION
PoFxRegisterPlugin	The PoFxRegisterPlugin routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPluginEx	The PoFxRegisterPluginEx routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterCoreDevice	The PoFxRegisterCoreDevice routine registers a new core system resource with the Windows power management framework (PoFx).
PoFxRegisterCrashdumpDevice	The PoFxRegisterCrashdumpDevice routine registers a crash-dump device.
PoFxPowerOnCrashdumpDevice	The PoFxPowerOnCrashdumpDevice routine requests that a crash-dump device be turned on.

CALLBACK FUNCTION	DESCRIPTION
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS	The EnumerateUnmaskedInterrupts routine enumerates interrupt sources whose interrupts are unmasked and enabled.
POFXCALLBACKPLATFORMIDLEVETO	The PlatformIdleVeto routine increments or decrements the veto count for a veto code for a platform idle state.
POFXCALLBACKPROCESSORHALT	The ProcessorHalt routine prepares the processor to be halted.
POFXCALLBACKPROCESSORIDLEVETO	The ProcessorIdleVeto routine increments or decrements the veto count for a veto code for a processor idle state.
POFXCALLBACKREQUESTCOMMON	The RequestCommon routine is a generic request handler.
POFXCALLBACKREQUESTINTERRUPT	The RequestInterrupt routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.
POFXCALLBACKREQUESTWORKER	The RequestWorker routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.
POFXCALLBACKCRITICALRESOURCE	The TransitionCriticalResource routine changes the active/inactive state of a core system component.
POFXCALLBACKUPDATEPLATFORMIDLESTATE	The UpdatePlatformIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.

CALLBACK FUNCTION	DESCRIPTION
POFXCALLBACKUPDATEPROCESSORIDLESTATE	The UpdateProcessorIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.
ComponentCriticalTransitionCallback	The ComponentCriticalTransitionCallback callback routine handles a transition of the specified component between the F0 (fully on) and low-power Fx component power states.

Configuration Manager Routines

The configuration manager routines use the CmXxx naming convention.

- CmCallbackGetKeyObjectID
- CmCallbackGetKeyObjectIDEx
- CmCallbackReleaseKeyObjectIDEx
- CmGetBoundTransaction
- CmGetCallbackVersion
- CmRegisterCallback
- CmRegisterCallbackEx
- CmSetCallbackObjectContext
- CmUnRegisterCallback

Kernel Transaction Manager (KTM) Routines

This section describes the routines, structures, and enumerations that the Kernel Transaction Manager (KTM) provides.

Transaction Manager Object Routines

This section includes the following topics:

- TmRecoverTransactionManager
- ZwCreateTransactionManager
- ZwOpenTransactionManager
- ZwQueryInformationTransactionManager
- ZwRecoverTransactionManager
- ZwRollforwardTransactionManager

Transaction Object Routines

This section includes the following topics:

- TmCommitTransaction
- TmGetTransactionId
- TmIsTransactionActive
- TmRollbackTransaction
- ZwCommitTransaction
- ZwCreateTransaction
- ZwEnumerateTransactionObject
- ZwOpenTransaction
- ZwQueryInformationTransaction
- ZwRollbackTransaction

- ZwSetInformationTransaction

Enlistment Object Routines

This section includes the following topics:

- TmCommitComplete
- TmCommitEnlistment
- TmCreateEnlistment
- TmDereferenceEnlistmentKey
- TmPrepareComplete
- TmPrePrepareComplete
- TmPrepareEnlistment
- TmPrePrepareEnlistment
- TmReadOnlyEnlistment
- TmRecoverEnlistment
- TmReferenceEnlistmentKey
- TmRequestOutcomeEnlistment
- TmRollbackComplete
- TmRollbackEnlistment
- TmSinglePhaseReject
- ZwCommitComplete
- ZwCommitEnlistment
- ZwCreateEnlistment
- ZwOpenEnlistment
- ZwPrepareComplete
- ZwPrePrepareComplete
- ZwPrepareEnlistment
- ZwPrePrepareEnlistment
- ZwQueryInformationEnlistment
- ZwReadOnlyEnlistment
- ZwRecoverEnlistment
- ZwRollbackComplete
- ZwRollbackEnlistment
- ZwSetInformationEnlistment
- ZwSinglePhaseReject

Resource Manager Object Routines

This section includes the following topics:

- ResourceManagerNotification
- TmEnableCallbacks
- TmRecoverResourceManager
- ZwCreateResourceManager
- ZwGetNotificationResourceManager
- ZwOpenResourceManager
- ZwQueryInformationResourceManager
- ZwRecoverResourceManager
- ZwSetInformationResourceManager

Security Reference Monitor Routines

Generally, higher-level drivers, particularly network drivers, call these routines.

References for the SeXxx routines are in alphabetical order.

- SeAccessCheck
- SeAssignSecurity
- SeAssignSecurityEx
- SeDeassignSecurity
- SeFreePrivileges
- ○ SeSinglePrivilegeCheck SeValidSecurityDescriptor

Core Kernel Library Support Routines

All kernel-mode drivers except video and SCSI miniport drivers and NDIS drivers are likely to call at least some KeXxx routines.

This section describes references for the KeXxx routines, in alphabetical order.

For an overview of the functionality of these routines, see Summary of Kernel-Mode Support Routines.

The following routines are reserved for system use:

- KeAcquireSpinLockRaiseToSynch
- KeBreakinBreakpoint
- KeEnterKernelDebugger
- KeFlushWriteBuffer
- KeGetBugMessageText
- KeRaiseIrqlToSynchLevel
- KeRemoveByKeyDeviceQueueIfBusy
- KeSetTimeUpdateNotifyRoutine

Executive Library Support Routines

This section describes the executive library support routines. These routines use the ExXxx naming convention and are listed in alphabetical order.

The following executive support routines are reserved for system use. Do not use them in your driver.

ROUTINE	REPLACEMENT
ExAcquireSpinLock	Use KeAcquireSpinLock instead.
ExAcquireSpinLockAtDpcLevel	Use KeAcquireSpinLockAtDpcLevel instead.
ExfInterlockedDecrementLong	Use InterlockedDecrement instead.
ExfInterlockedExchangeUlong	Use InterlockedExchange instead.
ExfInterlockedIncrementLong	Use InterlockedIncrement instead.
ExfInterlockedPopEntryList	Use ExInterlockedPopEntryList instead.

ROUTINE	REPLACEMENT
ExfInterlockedPushEntryList	Use ExInterlockedPushEntryList instead.
ExReleaseSpinLock	Use KeReleaseSpinLock instead.
ExReleaseSpinLockFromDpcLevel	Use KeReleaseSpinLockFromDpcLevel instead.
ExVerifySuite	

CLFS Library Routines

This section contains reference pages for routines implemented by the Common Log File System (CLFS). For a listing of the CLFS management routines, see CLFS Management Library Routines. For a conceptual discussion of CLFS, see Common Log File System in the design guide section of this documentation. For definitions of key terms used in the CLFS documentation, see CLFS Terminology.

CALLBACK FUNCTION	DESCRIPTION
ClfsAddLogContainer	The ClfsAddLogContainer routine adds a container to a CLFS log.
ClfsAddLogContainerSet	The ClfsAddLogContainerSet routine atomically adds a set of containers to a CLFS log.
ClfsAdvanceLogBase	The ClfsAdvanceLogBase routine sets the base LSN of a CLFS stream.
ClfsAlignReservedLog	The ClfsAlignReservedLog routine calculates the size of the space that must be reserved for a specified set of records. The size calculation includes the space required for headers and the space required for sector alignment.
ClfsAllocReservedLog	The ClfsAllocReservedLog routine reserves space in a marshalling area for a set of records.
ClfsCloseAndResetLogFile	The ClfsCloseAndResetLogFile routine releases all references to a specified log file object and marks its associated stream for reset.
ClfsCloseLogFileObject	The ClfsCloseLogFileObject routine releases all references to a log file object.
ClfsCreateLogFile	The ClfsCreateLogFile routine creates or opens a CLFS stream. If necessary, ClfsCreateLogFile also creates the underlying physical log that holds the stream's records.
ClfsCreateMarshallingArea	The ClfsCreateMarshallingArea routine creates a marshalling area for a CLFS stream and returns a pointer to an opaque context that represents the new marshalling area.
ClfsCreateScanContext	The ClfsCreateScanContext routine creates a scan context that can be used to iterate over the containers of a specified CLFS log.

CALLBACK FUNCTION	DESCRIPTION
ClfsDeleteLogByPointer	The ClfsDeleteLogByPointer routine marks a CLFS stream for deletion.
ClfsDeleteLogFile	The ClfsDeleteLogFile routine marks a CLFS stream for deletion.
ClfsDeleteMarshallingArea	The ClfsDeleteMarshallingArea routine deletes a marshalling area.
ClfsFlushBuffers	The ClfsFlushBuffers routine forces all log I/O blocks in a specified marshalling area to stable storage.
ClfsFlushToLsn	The ClfsFlushToLsn routine forces, to stable storage, all records that have an LSN less than or equal to a specified LSN.
ClfsGetContainerName	The ClfsGetContainerName routine returns the path name of a specified container.
ClfsGetIoStatistics	The ClfsGetIoStatistics routine returns I/O statistics for a specified CLFS log.
ClfsLsnBlockOffset	The ClfsLsnBlockOffset routine returns the sector-aligned block offset contained in a specified LSN.
ClfsLsnContainer	The ClfsLsnContainer routine returns the logical container identifier contained in a specified LSN.
ClfsLsnCreate	The ClfsLsnCreate routine creates a log sequence number (LSN), given a container identifier, a block offset, and a record sequence number.
ClfsLsnEqual	The ClfsLsnEqual routine determines whether two LSNs from the same stream are equal.
ClfsLsnGreater	The ClfsLsnGreater routine determines whether one LSN is greater than another LSN. The two LSNs must be from the same stream.
ClfsLsnLess	The ClfsLsnLess routine determines whether one LSN is less than another LSN. The two LSNs must be from the same stream.
ClfsLsnNull	The ClfsLsnNull routine determines whether a specified LSN is equal to the smallest possible LSN, CLFS_LSN_NULL.
ClfsLsnRecordSequence	The ClfsLsnRecordSequence routine returns the record sequence number contained in a specified LSN.
ClfsQueryLogFileInformation	The ClfsQueryLogFileInformation routine returns metadata and state information for a specified CLFS stream or its underlying physical log or both.

CALLBACK FUNCTION	DESCRIPTION
ClfsReadLogRecord	The ClfsReadLogRecord routine reads a target record in a CLFS stream and returns a read context that the caller can use to read the records preceding or following it in the stream.
ClfsReadNextLogRecord	The ClfsReadNextLogRecord routine reads the next record in a sequence, relative to the current record in a read context.
ClfsReadPreviousRestartArea	The ClfsReadPreviousRestartArea routine reads the previous restart record relative to the current record in a read context.
ClfsReadRestartArea	The ClfsReadRestartArea routine reads the restart record that was most recently written to a specified CLFS stream.
ClfsRemoveLogContainer	The ClfsRemoveLogContainer routine removes a container from a CLFS log.
ClfsRemoveLogContainerSet	The ClfsRemoveLogContainerSet routine atomically removes a set of containers from a CLFS log.
ClfsReserveAndAppendLog	The ClfsReserveAndAppendLog routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically.
ClfsReserveAndAppendLogAligned	The ClfsReserveAndAppendLogAligned routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically. The record's data is aligned on specified boundaries.
ClfsScanLogContainers	The ClfsScanLogContainers routine retrieves descriptive information for a sequence of containers that belong to a particular CLFS log.
ClfsSetArchiveTail	The ClfsSetArchiveTail routine sets the archive tail of a CLFS log to a specified LSN.
ClfsSetEndOfLog	The ClfsSetEndOfLog routine truncates a CLFS stream.
ClfsSetLogFileInformation	The ClfsSetLogFileInformation routine sets metadata and state information for a specified stream and its underlying physical log.
ClfsTerminateReadLog	The ClfsTerminateReadLog routine invalidates a specified read context after freeing resources associated with the context.
ClfsWriteRestartArea	The ClfsWriteRestartArea routine atomically appends a new restart record to a CLFS stream, flushes the restart record to stable storage, and optionally updates the base LSN of the stream.

IRPs

Microsoft Windows use I/O Request Packets (IRPs) to send messages to device drivers. An IRP is a data structure that contains specific information used to convey the status of an event. For more information about the IRP data structure, see IRP and IRP Major Function Codes.

Your driver can use System-Supplied Driver Interfaces to send IRPs to other drivers.

In addition to the Standard IRP Codes, there are three additional types of IRPs for specific technologies:

- Plug and Play IRPs, see Plug and Play Minor IRPs
- Power Management IRPs, see Power Management Minor IRPs
- Windows Management Instrumentation (WMI) IRPs, see WMI Minor IRPs

This section describes kernel-mode support routines that drivers can call:

- While processing IRPs.
- To allocate and set up IRPs for requests from higher-level drivers to lower drivers.
- To use file objects.

Processing IRPs

FUNCTION	DESCRIPTION
IoGetCurrentIrpStackLocation	Returns a pointer to the caller's I/O stack location in a given IRP.
IoGetNextIrpStackLocation	Returns a pointer to the next-lower-level driver's I/O stack location in a given IRP.
IoCopyCurrentIrpStackLocationToNext Copies the IRP stack parameters from the current stack location to the stack location of the next-lower driver and allows the current driver to set an I/O completion routine.	
IoSkipCurrentIrpStackLocation	Copies the IRP stack parameters from the current stack location to the stack location of the next-lower driver and does not allow the current driver to set an I/O completion routine.
IoGetRelatedDeviceObject	Returns a pointer to the device object represented by a given file object.
IoGetFunctionCodeFromCtlCode	Returns the value of the function field within a given IOCTL_XXX or FSCTL_XXX code.
IoValidateDeviceIoControlAccess	Verifies that the sender of an IRP_MJ_DEVICE_CONTROL or IRP_MJ_FILE_SYSTEM_CONTROL IRP has the access specified.
IoSetCompletionRoutine	Registers a driver-supplied IoCompletion routine for an IRP, so the IoCompletion routine is called when the next-lower-level driver has completed the requested operation in one or more of the following ways: successfully, with an error, or by canceling the IRP.
IoSetCompletionRoutineEx	Same as IoSetCompletionRoutine, except that it guarantees a non-Plug and Play driver is not unloaded before the IoCompletion routine exits.
IoCallDriver	Sends an IRP to a lower-level driver.
PoCallDriver	Sends an IRP with major function code IRP_MJ_POWER to the next-lower driver.

FUNCTION	DESCRIPTION
IoForwardIrpSynchronously	Sends an IRP to a lower-level driver synchronously.
IoMarkIrpPending	Marks a given IRP indicating that STATUS_PENDING was returned because further processing is required by another driver routine or by a lower-level driver.
IoStartPacket	Calls the driver's StartIo routine with the given IRP for the given device object or inserts the IRP into the device queue if the device is already busy, specifying whether the IRP is cancelable.
IoSetStartIoAttributes	Sets attributes for when the driver's StartIo routine will execute.
IoAcquireCancelSpinLock	Synchronizes cancelable state transitions for IRPs in a multiprocessor-safe manner.
IoSetCancelRoutine	Sets or clears the Cancel routine in an IRP. Setting a Cancel routine makes an IRP cancelable.
IoReleaseCancelSpinLock	Releases the cancel spin lock when the driver has changed the cancelable state of an IRP or releases the cancel spin lock from the driver's Cancel routine.
IoCancelIrp	Marks an IRP as canceled.
IoReadPartitionTable	Returns a list of partitions on a disk with a given sector size.
IoSetPartitionInformation	Sets the partition type and number for a (disk) partition.
IoWritePartitionTable	Writes partition tables for a disk, given the device object representing the disk, the sector size, and a pointer to a buffer containing the drive geometry.
IoAllocateErrorLogEntry	Allocates and initializes an error log packet; returns a pointer so that the caller can supply error-log data and call IoWriteErrorLogEntry with the packet.
IoWriteErrorLogEntry	Queues a previously allocated and filled-in error log packet to the system error logging thread.
IoIsErrorUserInduced	Returns a Boolean value indicating whether an I/O request failed due to one of the following conditions: STATUS_IO_TIMEOUT, STATUS_DEVICE_NOT_READY, STATUS_UNRECOGNIZED_MEDIA, STATUS_VERIFY_REQUIRED, STATUS_WRONG_VOLUME, STATUS_MEDIA_WRITE_PROTECTED, or STATUS_NO_MEDIA_IN_DEVICE. If the result is TRUE, a removable-media driver must call IoSetHardErrorOrVerifyDevice before completing the IRP.

FUNCTION	DESCRIPTION
IoSetHardErrorOrVerifyDevice	Supplies the device object for which the given IRP was failed due to a user-induced error, such as supplying the incorrect media for the requested operation or changing the media before the requested operation was completed. A file system driver uses the associated device object to notify the user, who can then correct the error or retry the operation.
IoRaiseHardError	Notifies the user that the given IRP was failed on the given device object for an optional VPB, so that the user can correct the error or retry the operation.
IoRaiseInformationalHardError	Notifies the user of an error, providing an I/O error status and an optional string supplying more information.
ExRaiseStatus	Raises an error status and causes a caller-supplied structured exception handler to be called. Useful only to highest-level drivers that supply exception handlers, in particular to file systems.
IoStartNextPacket	Dequeues the next IRP for a given device object, specifies whether the IRP is cancelable, and calls the driver's StartIo routine.
IoStartNextPacketByKey	Dequeues the next IRP for a device object according to a specified sort-key value, specifies whether the IRP is cancelable, and calls the driver's StartIo routine.
IoCompleteRequest	Completes an I/O request, giving a priority boost to the original caller and returning a given IRP to the I/O system for disposal: either to call any IoCompletion routines supplied by higher-level drivers, or to return status to the original requester of the operation.
IoGetCurrentProcess	Returns a pointer to the current process. Useful only to highest-level drivers.
IoGetInitialStack	Returns the initial base address of the current thread's stack. Useful only to highest-level drivers.
IoGetRemainingStackSize	Returns the amount of available stack space. Useful only to highest-level drivers.
IoGetStackLimits	Returns the boundaries of the current thread's stack frame. Useful only to highest-level drivers.
IoCsqInitialize	Initializes the dispatch table for a driver's cancel-safe IRP queue.
IoCsqInsertIrp	Inserts an IRP into a driver's cancel-safe IRP queue.
IoCsqRemoveIrp	Removes the specified IRP from a driver's cancel-safe IRP queue.
IoCsqRemoveNextIrp	Removes the next IRP from a driver's cancel-safe IRP queue.

Driver-Allocated IRPs

FUNCTION	DESCRIPTION
oBuildAsynchronousFsdRequest	Allocates and sets up an IRP that specifies a major function code (IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_SHUTDOWN, or IRP_MJ_FLUSH_BUFFERS) with a pointer to: The lower driver's device object on which the I/O should occur. A buffer that will contain the data to be read or that contains the data to be written. The length of the buffer in bytes. The starting offset on the media. The I/O status block where the called driver can return status information and the caller's IoCompletion routine can access it. Returns a pointer to the IRP so the caller can set any necessary minor function code and set up its IoCompletion routine before sending the IRP to the target driver.
IoBuildSynchronousFsdRequest	Allocates and sets up an IRP specifying a major function code (IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_SHUTDOWN, or IRP_MJ_FLUSH_BUFFERS) with a pointer to: The lower driver's device object on which the I/O should occur. A buffer that will contain the data to be read or that contains the data to be written. The length of the buffer in bytes. The starting offset on the media. An event object to be set to the Signaled state when the requested operation completes. The I/O status block where the called driver can return status information and the caller's IoCompletion routine can access it. Returns a pointer to the IRP so the caller can set any necessary minor function code and set up its IoCompletion routine before sending the IRP to the target driver.
IoBuildDeviceIoControlRequest	Allocates and sets up an IRP specifying a major function code (either IRP_MJ_INTERNAL_DEVICE_CONTROL or IRP_MJ_DEVICE_CONTROL) with an optional input or output buffer; a pointer to the lower driver's device object; an event to be set to the Signaled state when the requested operation completes; and an I/O status block to be set by the driver that receives the IRP. Returns a pointer to the IRP so the caller can set the appropriate IOCTL_XXX before sending the IRP to the next-lower-level driver.
PoRequestPowerIrp	Allocates and initializes an IRP with major function code IRP_MJ_POWER and then sends the IRP to the top-level driver in the device stack for the specified device object.
IoSizeOfIrp	Returns the size in bytes required for an IRP with a given count of I/O stack locations.
IoAllocateIrp	Allocates an IRP, given the number of I/O stack locations (optionally, for the caller, but at least one for each driver layered under the caller) and whether to charge quota against the caller. Returns a pointer to an IRP in nonpaged system space if successful; otherwise, returns NULL.
IoInitializeIrp	Initializes an IRP, given a pointer to an already allocated IRP, its length in bytes, and its number of I/O stack locations.

FUNCTION	DESCRIPTION
IoSetNextIrpStackLocation	Sets the current IRP stack location to the caller's location in an IRP. The stack location must have been allocated by a preceding call to IoAllocateIrp that specified a stack-size argument large enough to give the caller its own stack location.
IoAllocateMdl	Allocates an MDL large enough to map the starting address and length supplied by the caller; optionally associates the MDL with a given IRP.
IoBuildPartialMdl	Builds an MDL for the specified starting virtual address and length in bytes from a given source MDL. Drivers that split large transfer requests into a number of smaller transfers can call this routine.
IoFreeMdl	Releases a given MDL allocated by the caller.
IoMakeAssociatedIrp	Allocates and initializes an IRP to be associated with a master IRP sent to the highest-level driver, allowing the driver to "split" the original request and send associated IRPs on to lower-level drivers or to the device.
IoSetCompletionRoutine	Registers a driver-supplied IoCompletion routine with a given IRP, so that the IoCompletion routine is called when lower-level drivers have completed the request. The IoCompletion routine lets the caller release the IRP it allocated with IoAllocateIrp or IoBuildAsynchronousFsdRequest; to release any other resources it allocated to set up an IRP for lower drivers; and to perform any I/O completion processing necessary.
IoSetCompletionRoutineEx	Same as IoSetCompletionRoutine, except that it guarantees a non-Plug and Play driver is not unloaded before the IoCompletion routine exits.
IoCallDriver	Sends an IRP to a lower-level driver.
IoFreeIrp	Releases an IRP that was allocated by the caller.
IoReuseIrp	Reinitializes for reuse an IRP that was previously allocated by IoAllocateIrp.

File Objects

FUNCTION	DESCRIPTION
nitializeObjectAttributes	Initializes a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a ZwCreateXxx or ZwOpenXxx routine.
ZwCreateFile	Creates or opens a file object representing a physical, logical, or virtual device, a directory, a data file, or a volume.
ZwQueryInformationFile	Returns information about the state or attributes of an open file.

FUNCTION	DESCRIPTION
IoGetFileObjectGenericMapping	Returns information about the mapping between generic access rights and specific access rights for file objects.
ZwReadFile	Returns data from an open file.
ZwSetInformationFile	Changes information about the state or attributes of an open file.
ZwWriteFile	Transfers data to an open file.
ZwClose	Releases the handle for an opened object, causing the handle to become invalid and decrementing the reference count of the object handle.

Silo DDIs

This section describes the silo DDIs. These DDIs offer the ability for kernel components to learn about the server silos that are created and destroyed on a machine. Components register to receive notifications for these events and optionally store state that is associated with each silo.

Context management

These DDIs provide the ability to assign and retrieve context structures on silo objects. This allows drivers to attach per-silo information for each silo.

- PsAllocSiloContextSlot
- PsFreeSiloContextSlot
- PsCreateSiloContext
- PsInsertSiloContext
- PsReplaceSiloContext
- PsInsertPermanentSiloContext
- PsGetPermanentSiloContext
- PsMakeSiloContextPermanent
- PsGetSiloContext
- PsRemoveSiloContext
- PsReferenceSiloContext
- PsDereferenceSiloContext
- SILO_CONTEXT_CLEANUP_CALLBACK

Threading

These DDIs provide the ability to set and retrieve the silo for the current thread.

- PsAttachSiloToCurrentThread
- PsDetachSiloFromCurrentThread
- PsGetCurrentSilo
- PsGetCurrentServerSilo

Monitoring

These DDIs provide the ability for a driver to receive notifications about silo creation and termination events.

- PsRegisterSiloMonitor

- PsUnregisterSiloMonitor
- PsStartSiloMonitor
- PsGetSiloMonitorContextSlot
- SILO_MONITOR_CREATE_CALLBACK
- SILO_MONITOR_TERMINATE_CALLBACK

Helpers

These DDIs are helpful for working with silo objects.

- PsGetJobSilo
- PsGetJobServerSilo
- PsGetEffectiveServerSilo
- PsIsHostSilo
- PsGetHostSilo
- PsTerminateServerSilo

Synchronization

This section describes the kernel-mode support routines that drivers can call to:

- Synchronize the execution of their own standard driver routines (Driver Routines and I/O Objects).
- Temporarily change the current IRQL for a call to a support routine or that return the current IRQL (IRQL).
- Synchronize access to resources with spin locks or to perform interlocked operations without spin locks (Spin Locks and Interlocks).
- Manage time-outs or determine system time (Timers).
- Use system threads or to manage synchronization within a nonarbitrary thread context (Driver Threads, Dispatcher Objects, and Resources).

Driver Routines and I/O Objects

FUNCTION	DESCRIPTION
KeSynchronizeExecution	Synchronizes the execution of a driver-supplied SynchCritSection routine with that of the ISR associated with a set of interrupt objects, given a pointer to the interrupt objects.
IoRequestDpc	Queues a driver-supplied DpcForIsr routine to complete interrupt-driven I/O processing at a lower IRQL.
KeInsertQueueDpc	Queues a DPC to be executed as soon as the IRQL of a processor drops below DISPATCH_LEVEL; returns FALSE if the DPC object is already queued.
KeRemoveQueueDpc	Removes a given DPC object from the DPC queue; returns FALSE if the object is not in the queue.
KeSetImportanceDpc	Controls how a particular DPC is queued and, to some degree, how soon the DPC routine is run.
KeSetTargetProcessorDpc	Controls on which processor a particular DPC subsequently will be queued.
KeFlushQueuedDpcs	Call this routine to wait for all queued DPCs to execute.

FUNCTION	DESCRIPTION
AllocateAdapterChannel	Connects a device object to an adapter object and calls a driver-supplied AdapterControl routine to carry out an I/O operation through the system DMA controller or a bus-master adapter as soon as the appropriate DMA channel and any necessary map registers are available. (This routine reserves exclusive access to a DMA channel and map registers for the specified device.)
FreeAdapterChannel	Releases an adapter object, representing a system DMA channel, and optionally releases map registers, if any were allocated.
FreeMapRegisters	Releases a set of map registers that were saved from a call to AllocateAdapterChannel, after the registers have been used by IoMapTransfer and the bus-master DMA transfer is complete.
IoAllocateController	Connects a device object to a controller object and calls a driver-supplied ControllerControl routine to carry out an I/O operation on the device controller as soon as the controller is not busy. (This routine reserves exclusive access to the hardware controller for the specified device.)
IoFreeController	Releases a controller object, provided that all device operations queued to the controller for the current IRP have completed.
IoStartTimer	Enables the timer for a given device object and calls the driver-supplied IoTimer routine once per second thereafter.
IoStopTimer	Disables the timer for a given device object so that the driver-supplied IoTimer routine is not called unless the driver reenables the timer.
KeSetTimer	Sets the absolute or relative interval at which a timer object will be set to the Signaled state and optionally supplies a timer DPC to be executed after the interval expires.
KeSetTimerEx	Sets the absolute or relative interval at which a timer object will be set to the Signaled state, optionally supplies a timer DPC to be executed when the interval expires, and optionally supplies a recurring interval for the timer.
KeCancelTimer	Cancels a timer object before the interval passed to KeSetTimer expires; dequeues a timer DPC before the timer interval, if any was set, expires.
KeReadStateTimer	Returns whether a given timer object is set to the Signaled state.
IoStartPacket	Calls the driver's StartIo routine with the given IRP for the given device object or inserts the IRP into the device queue if the device is already busy, specifying whether the IRP is cancelable.
IoStartNextPacket	Dequeues the next IRP for a given device object, specifying whether the IRP is cancelable, and calls the driver's StartIo routine.

FUNCTION	DESCRIPTION
IoStartNextPacketByKey	Dequeues the next IRP, according to the specified sort-key value, for a given device object. Specifies whether the IRP is cancelable and calls the driver's StartIo routine.
IoSetCompletionRoutine	Registers a driver-supplied IoCompletion routine with a given IRP, so the IoCompletion routine is called when the next-lower-level driver has completed the requested operation in one or more of the following ways: successfully, with an error, or by canceling the IRP.
IoSetCompletionRoutineEx	Same as IoSetCompletionRoutine, except that it guarantees a non-Plug and Play driver is not unloaded before the IoCompletion routine exits.
IoSetCancelRoutine	Sets or clears the Cancel routine in an IRP. Setting a Cancel routine makes an IRP cancelable.
KeStallExecutionProcessor	Stalls the caller (a device driver) for a given interval on the current processor.
ExAcquireResourceExclusiveLite	Acquires an initialized resource for exclusive access by the calling thread and optionally waits for the resource to be acquired.
ExAcquireResourceSharedLite	Acquires an initialized resource for shared access by the calling thread and optionally waits for the resource to be acquired.
ExAcquireSharedStarveExclusive	Acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.
ExAcquireSharedWaitForExclusive	Acquires a given resource for shared access, optionally waiting for any pending exclusive waiters to acquire and release the resource first.
ExReleaseResourceForThreadLite	Releases a given resource that was acquired by the given thread.
ZwReadFile	Reads data from an open file. If the caller opened the file object with certain parameters, the caller can wait for the file handle to be returned for completion of the I/O.
ZwWriteFile	Writes data to an open file. If the caller opened the file object with certain parameters, the caller can wait for the file handle to be returned for completion of the I/O.

IRQL

FUNCTION	DESCRIPTION
KeRaiseIrql	Raises the hardware priority to a given IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

FUNCTION	DESCRIPTION
KeRaiseIrqlToDpcLevel	aises the hardware priority to IRQL DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeLowerIrql	Restores the IRQL on the current processor to its original value.
KeGetCurrentIrql	Returns the current hardware priority IRQL value.

Spin Locks and Interlocks

FUNCTION	DESCRIPTION
oAcquireCancelSpinLock	Synchronizes cancelable state transitions for IRPs in a multiprocessor-safe manner.
IoSetCancelRoutine	Sets or clears the Cancel routine in an IRP during a cancelable state transition. Setting a Cancel routine makes an IRP cancelable.
IoReleaseCancelSpinLock	Releases the cancel spin lock when the driver has changed the cancelable state of an IRP or releases the cancel spin lock from the driver's Cancel routine.
KeInitializeSpinLock	Initializes a variable of type KSPIN_LOCK, used to synchronize access to data shared among non-ISR routines. An initialized spin lock also is a required parameter to the ExInterlockedXxx routines.
KeAcquireSpinLock	Acquires a spin lock so the caller can synchronize access to shared data safely on multiprocessor platforms.
KeAcquireSpinLockRaiseToDpc	Acquires a spin lock so the caller can synchronize access to shared data safely on multiprocessor platforms.
KeReleaseSpinLock	Releases a spin lock that was acquired by calling KeAcquireSpinLock and restores the original IRQL at which the caller was running.
KeAcquireSpinLockAtDpcLevel	Acquires a spin lock, provided that the caller is already running at IRQL DISPATCH_LEVEL.
KeTryToAcquireSpinLockAtDpcLevel	Acquires a spin lock that is not already held, provided that the caller is already running at IRQL DISPATCH_LEVEL.
KeReleaseSpinLockFromDpcLevel	Releases a spin lock that was acquired by calling KeAcquireSpinLockAtDpcLevel.
KeAcquireInStackQueuedSpinLock	Acquires a queued spin lock so the caller can synchronize access to shared data safely on multiprocessor platforms.
KeReleaseInStackQueuedSpinLock	Releases a queued spin lock that was acquired by calling KeAcquireInStackQueuedSpinLock.

FUNCTION	DESCRIPTION
KeAcquireInStackQueuedSpinLockAtDpcLevel	Acquires a queued spin lock, provided that the caller is already running at IRQL DISPATCH_LEVEL.
KeReleaseInStackQueuedSpinLockFromDpcLevel	Releases a queued spin lock that was acquired by calling KeAcquireInStackQueuedSpinLockAtDpcLevel.
KeAcquireInterruptSpinLock	Acquires the spin lock that synchronizes access with an interrupt's ISR.
KeReleaseInterruptSpinLock	Release the spin lock that synchronized access with an interrupt's ISR.
ExInterlockedXxxList	Insert and remove IRPs in a driver-managed internal queue, which is protected by an initialized spin lock for which the driver provides the storage.
KeXxxDeviceQueue	Insert and remove IRPs in a driver-allocated and managed internal device queue object, which is protected by a built-in spin lock.
ExInterlockedAddUlong	Adds a value to a variable of type ULONG as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the variable; returns the value of the variable before the call occurred.
ExInterlockedAddLargeInteger	Adds a value to a variable of type LARGE_INTEGER as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the variable; returns the value of the variable before the call occurred.
InterlockedIncrement	Increments a variable of type LONG as an atomic operation. The sign of the return value is the sign of the result of the operation.
InterlockedDecrement	Decrements a variable of type LONG as an atomic operation. The sign of the return value is the sign of the result of the operation.
InterlockedExchange	Sets a variable of type LONG to a specified value as an atomic operation; returns the value of the variable before the call occurred.
InterlockedExchangeAdd	Adds a value to a given integer variable as an atomic operation; returns the value of the variable before the call occurred.
InterlockedCompareExchange	Compares the values referenced by two pointers. If the values are equal, resets one of the values to a caller-supplied value in an atomic operation.
InterlockedCompareExchangePointer	Compares the pointers referenced by two pointers. If the pointer values are equal, resets one of the values to a caller-supplied value in an atomic operation.

FUNCTION	DESCRIPTION
ExInterlockedCompareExchange64	Compares one integer variable to another and, if they are equal, resets the first variable to a caller-supplied ULONGLONG-type value as an atomic operation.
KeGetCurrentProcessorNumber	Returns the current processor number when debugging spin lock usage in SMP machines.

Timers

FUNCTION	DESCRIPTION
oInitializeTimer	Associates a timer with the given device object and registers a driver-supplied IoTimer routine for the device object.
IoStartTimer	Enables the timer for a given device object and calls the driver-supplied IoTimer routine once every second.
IoStopTimer	Disables the timer for a given device object so the driver-supplied IoTimer routine is not called unless the driver reenables the timer.
KeInitializeDpc	Initializes a DPC object and sets up a driver-supplied CustomTimerDpc routine that can be called with a given context.
KeInitializeTimer	Initializes a notification timer object to the Not-Signaled state.
KeInitializeTimerEx	Initializes a notification or synchronization timer object to the Not-Signaled state.
KeSetTimer	Sets the absolute or relative interval at which a timer object will be set to the Signaled state; optionally supplies a timer DPC to be executed when the interval expires.
KeSetTimerEx	Sets the absolute or relative interval at which a timer object will be set to the Signaled state; optionally supplies a timer DPC to be executed when the interval expires; and optionally supplies a recurring interval for the timer.
KeCancelTimer	Cancels a timer object before the interval passed to KeSetTimer expires; dequeues a timer DPC before the timer interval, if any was set, expires.
KeReadStateTimer	Returns TRUE if a given timer object is set to the Signaled state.
KeQuerySystemTime	Returns the current system time.
KeQueryRuntimeThread	Returns the accumulated kernel-mode and user-mode run time.
KeQueryTickCount	Returns the number of interval-timer interrupts that have occurred since the system was booted.

FUNCTION	DESCRIPTION
KeQueryTimeIncrement	Returns the number of 100-nanosecond units that are added to the system time at each interval-timer interrupt.
KeQueryInterruptTime	Returns the current value of the system interrupt-time count in 100-nanosecond units, with accuracy to within system clock tick.
KeQueryInterruptTimePrecise	Returns the current value of the system interrupt-time count in 100-nanosecond units, with accuracy to within a microsecond.
KeQueryPerformanceCounter	Returns the system performance counter value in hertz.

Driver Threads, Dispatcher Objects, and Resources

FUNCTION	DESCRIPTION
KeDelayExecutionThread	Puts the current thread into an alertable or nonalertable wait state for a given interval.
ExInitializeResourceLite	Initializes a resource, for which the caller provides the storage, to be used for synchronization by a set of threads (shared readers, exclusive writers).
ExReinitializeResourceLite	Reinitializes an existing resource variable.
ExAcquireResourceExclusiveLite	Acquires an initialized resource for exclusive access by the calling thread and optionally waits for the resource to be acquired.
ExAcquireResourceSharedLite	Acquires an initialized resource for shared access by the calling thread and optionally waits for the resource to be acquired.
ExAcquireSharedStarveExclusive	Acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.
ExAcquireSharedWaitForExclusive	Acquires a given resource for shared access, optionally waiting for any pending exclusive waiters to acquire and release the resource first.
ExIsResourceAcquiredExclusiveLite	Returns whether the calling thread has exclusive access to a given resource.
ExIsResourceAcquiredSharedLite	Returns how many times the calling thread has acquired shared access to a given resource.
ExGetExclusiveWaiterCount	Returns the number of threads currently waiting to acquire a given resource for exclusive access.
ExGetSharedWaiterCount	Returns the number of threads currently waiting to acquire a given resource for shared access.

FUNCTION	DESCRIPTION
ExConvertExclusiveToSharedLite	Converts a given resource from acquired for exclusive access to acquired for shared access.
ExGetCurrentResourceThread	Returns the thread ID of the current thread.
ExReleaseResourceForThreadLite	Releases a given resource that was acquired by the given thread.
ExDeleteResourceLite	Deletes a caller-initialized resource from the system's resource list.
IoQueueWorkItem	Queues an initialized work queue item so the driver-supplied routine will be called when a system worker thread is given control.
KeSetTimer	Sets the absolute or relative interval at which a timer object will be set to the Signaled state, and optionally supplies a timer DPC to be executed when the interval expires.
KeSetTimerEx	Sets the absolute or relative interval at which a timer object will be set to the Signaled state. Optionally supplies a timer DPC to be executed when the interval expires and a recurring interval for the timer.
KeCancelTimer	Cancels a timer object before the interval passed to KeSetTimer expires. Dequeues a timer DPC before the timer interval (if any) expires.
KeReadStateTimer	Returns TRUE if a given timer object is set to the Signaled state.
KeSetEvent	Returns the previous state of a given event object and sets the event (if not already Signaled) to the Signaled state.
KeClearEvent	Resets an event to the Not-Signaled state.
KeResetEvent	Returns the previous state of an event object and resets the event to the Not-Signaled state.
KeReadStateEvent	Returns the current state (nonzero for Signaled or zero for Not-Signaled) of a given event object.
ExAcquireFastMutex	Acquires an initialized fast mutex, possibly after putting the caller into a wait state until it is acquired, and gives the calling thread ownership with APCs disabled.
ExTryToAcquireFastMutex	Acquires the given fast mutex immediately for the caller with APCs disabled, or returns FALSE.
ExReleaseFastMutex	Releases ownership of a fast mutex that was acquired with ExAcquireFastMutex or ExTryToAcquireFastMutex.
ExAcquireFastMutexUnsafe	Acquires an initialized fast mutex, possibly after putting the caller into a wait state until it is acquired.

FUNCTION	DESCRIPTION
ExReleaseFastMutexUnsafe	Releases ownership of a fast mutex that was acquired with ExAcquireFastMutexUnsafe.
KeReleaseMutex	Releases a given mutex object, specifying whether the caller will call one of the KeWaitXxx routines as soon as KeReleaseMutex returns the previous value of the mutex state (a zero for Signaled; otherwise, Not-Signaled).
KeReadStateMutex	Returns the current state (one for Signaled or any other value for Not-Signaled) of a given mutex object.
KeReleaseSemaphore	Releases a given semaphore object. Supplies a (run-time) priority boost for waiting threads if the release sets the semaphore to the Signaled state. Augments the semaphore count by a given value and specifies whether the caller will call one of the KeWaitXxx routines as soon as KeReleaseSemaphore returns.
KeReadStateSemaphore	Returns the current state (zero for Not-Signaled or a positive value for Signaled) of a given semaphore object.
KeWaitForSingleObject	Puts the current thread into an alertable or nonalertable wait state until a given dispatcher object is set to the Signaled state or (optionally) until the wait times out.
KeWaitForMutexObject	Puts the current thread into an alertable or nonalertable wait state until a given mutex is set to the Signaled state or (optionally) until the wait times out.
KeWaitForMultipleObjects	Puts the current thread into an alertable or nonalertable wait state until any one or all of a number of dispatcher objects are set to the Signaled state or (optionally) until the wait times out.
PsGetCurrentThread	Returns a pointer to the current thread.
KeGetCurrentThread	Returns a pointer to the opaque thread object that represents the current thread.
IoGetCurrentProcess	Returns a pointer to the process of the current thread.
PsGetCurrentProcess	Returns a pointer to the process of the current thread.
KeEnterCriticalRegion	Temporarily disables the delivery of normal kernel APCs while a highest-level driver is running in the context of the user-mode thread that requested the current I/O operation. Special kernel-mode APCs are still delivered.
KeLeaveCriticalRegion	Reenables, as soon as possible, the delivery of normal kernel-mode APCs that were disabled by a preceding call to KeEnterCriticalRegion.
KeAreApcsDisabled	Returns TRUE if normal kernel-mode APCs are disabled.

FUNCTION	DESCRIPTION
KeSaveFloatingPointState	Saves the current thread's nonvolatile floating-point context so that the caller can carry out its own floating-point operations.
KeRestoreFloatingPointState	Restores the previous nonvolatile floating-point context that was saved with KeSaveFloatingPointState.
ZwSetInformationThread	Sets the priority of a given thread for which the caller has a handle.
PsGetCurrentProcessId	Returns the system-assigned identifier of the current process.
PsGetCurrentThreadId	Returns the system-assigned identifier of the current thread.
PsSetCreateProcessNotifyRoutine	Registers a highest level driver's callback routine that is subsequently executed whenever a new process is created or existing process deleted.
PsSetCreateThreadNotifyRoutine	Registers a highest level driver's callback routine that is subsequently executed whenever a new thread is created or an existing thread is deleted.
PsSetLoadImageNotifyRoutine	Registers a callback routine for a highest level system-profiling driver. The callback routine is subsequently executed whenever a new image is loaded for execution.

Memory Allocation and Buffer Management

This section describes the Windows kernel routines and macros that kernel-mode drivers call to allocate memory and manage I/O buffers.

The Windows memory manager provides a set of routines that kernel-mode drivers use to allocate and manage memory. These routines have names that begin with the prefix Mm.

This section contains reference pages for the MmXxx routines and memory-management macros. These reference pages are listed in alphabetical order.

For an overview of the functionality of these routines and macros, see Memory Allocation and Buffer Management. For an introduction to memory management support for kernel-mode drivers, see Memory Management for Windows Drivers.

The following routines are reserved for system use. Do not use them in your driver.

- MmAddPhysicalMemory
- MmAddPhysicalMemoryEx
- MmAddVerifierThunks
- MmCreateMirror
- MmGetMdlBaseVa
- MmGetPhysicalMemoryRanges
- MmGetProcedureAddress
- MmGetVirtualForPhysical
- MmIsVerifierEnabled
- MmIsIoSpaceActive

- MmMapUserAddressesToPage
- MmMapVideoDisplay
- MmMapViewInSessionSpace
- MmMapViewInSystemSpace
- MmMarkPhysicalMemoryAsBad
- MmMarkPhysicalMemoryAsGood
- MmProbeAndLockProcessPages
- MmRemovePhysicalMemory
- MmRemovePhysicalMemoryEx
- MmRotatePhysicalView
- MmUnmapVideoDisplay
- MmUnmapViewInSessionSpace
- MmUnmapViewInSystemSpace

For more information about memory allocation and buffer management, see Memory Management for Windows Drivers.

Buffer Management

The short-term buffer management routines are called by kernel-mode drivers to allocate and free temporary buffers.

FUNCTION	DESCRIPTION
ExAllocatePoolWithTag	Allocates (optionally cache-aligned) pool memory from paged or nonpaged system space. The caller-supplied tag is put into any crash dump of memory that occurs.
ExAllocatePoolWithQuotaTag	Allocates pool memory, charging quota against the original requester of the I/O operation. The caller-supplied tag is put into any crash dump of memory that occurs. Only highest-level drivers can call this routine.
ExFreePool	Releases memory to paged or nonpaged system space.
ExFreePoolWithTag	Releases memory with the specified pool tag.
ExInitializeNPagedLookasideList	Initializes a lookaside list of nonpaged memory. After successful initialization of the list, fixed-size blocks can be allocated from, and freed to, the lookaside list.
ExAllocateFromNPagedLookasideList	Removes the first entry from the specified lookaside list in nonpaged memory. If the lookaside list is empty, allocates an entry from nonpaged pool.
ExFreeToNPagedLookasideList	Returns an entry to the specified lookaside list in nonpaged memory. If the list has reached its maximum size, returns the entry to nonpaged pool.
ExDeleteNPagedLookasideList	Deletes a nonpaged lookaside list.
ExInitializePagedLookasideList	Initializes a lookaside list of paged memory. After successful initialization of the list, fixed-size blocks can be allocated from and freed to the lookaside list.

FUNCTION	DESCRIPTION
ExAllocateFromPagedLookasideList	Removes the first entry from the specified lookaside list in paged memory. If the lookaside list is empty, allocates an entry from paged pool.
ExFreeToPagedLookasideList	Returns an entry to the specified lookaside list in paged memory. If the list has reached its maximum size, returns the entry to paged pool.
ExDeletePagedLookasideList	Deletes a paged lookaside list.
MmQuerySystemSize	Returns an estimate (small, medium, or large) of the amount of memory available on the current platform.
MmIsThisAnNtAsSystem	Returns TRUE if the machine is running as a server. If this routine returns TRUE, the caller is likely to require more resources to process I/O requests, and the machine is a server so it is likely to have more resources available.

Long-Term Internal Driver Buffers

The long-term buffer management routines are called by kernel-mode drivers to allocate long-term driver-internal buffers.

FUNCTION	DESCRIPTION
MmAllocateContiguousMemory	Allocates a range of physically contiguous, cache-aligned memory in nonpaged pool.
MmFreeContiguousMemory	Releases a range of physically contiguous memory when the driver unloads.
MmAllocateNonCachedMemory	Allocates a virtual address range of noncached and cache-aligned memory in nonpaged system space (pool).
MmFreeNonCachedMemory	Releases a virtual address range of noncached memory in nonpaged system space when the driver unloads.
MmAllocateMappingAddress	Reserves a range of virtual address space that can later be mapped with MmMapLockedPagesWithReservedMapping.
MmFreeMappingAddress	Frees a reserved memory address reserved by MmAllocateMappingAddress.
AllocateCommonBuffer	Allocates and maps a logically contiguous region of memory that is simultaneously accessible both from the processor and from a device, given access to an adapter object, the requested length of the memory region to allocate, and access to variables where the starting logical and virtual addresses of the allocated region are returned. Returns TRUE if the requested length was allocated. Can be used for continuous bus-master DMA or for system DMA using the autoinitialize mode of a system DMA controller.

FUNCTION	DESCRIPTION
FreeCommonBuffer	Releases an allocated common buffer and unmaps it, given access to the adapter object, the length, and the starting logical and virtual addresses of the region to be freed when the driver unloads. Arguments must match those passed in the call to AllocateCommonBuffer.

Buffered Data and Buffer Initialization

The buffered-data and buffer-initialization routines are called by kernel-mode drivers to manage buffered data or initialize driver-allocated buffers.

FUNCTION	DESCRIPTION
RtlCompar	Memory
RtlCopyMemory	Copies the data from one caller-supplied buffer to another, given pointers to both buffers and the length in bytes to be copied.
RtlMoveMemory	Copies the data from one caller-supplied memory range to another, given pointers to the base of both ranges and the length in bytes to be copied.
RtlFillMemory	Fills a caller-supplied buffer with the specified UCHAR value, given a pointer to the buffer and the length in bytes to be filled.
RtlZeroMemory	Fills a buffer with zeros, given a pointer to the caller-supplied buffer and the length in bytes to be filled.
RtlStoreUshort	Stores a USHORT value at a given address, avoiding alignment faults.
RtlRetrieveUshort	Retrieves a USHORT value at a given address, avoiding alignment faults, and stores the value at a given address that is assumed to be aligned.
RtlStoreUlong	Stores a ULONG value at a given address, avoiding alignment faults.
RtlRetrieveUlong	Retrieves a ULONG value at a given address, avoiding alignment faults, and stores the value at a given address that is assumed to be aligned.

Address Mappings and MDLs

The address-mapping and MDL-management routines are called by kernel-mode drivers to manage address mappings and memory descriptor lists (MDLs).

FUNCTION	DESCRIPTION
oAllocateMdl	Allocates an MDL large enough to map the starting address and length supplied by the caller; optionally associates the MDL with a given IRP.

FUNCTION	DESCRIPTION
IoBuildPartialMdl	Builds an MDL for the specified starting virtual address and length in bytes from a given source MDL. Drivers that split large transfer requests into a number of smaller transfers can call this routine.
IoFreeMdl	Releases a given MDL allocated by the caller.
MmAllocatePagesForMdlEx	Allocates nonpaged, physical memory pages for an MDL.
MmBuildMdlForNonPagedPool	Fills in the corresponding physical addresses of a given MDL that specifies a range of virtual addresses in nonpaged pool.
MmCreateMdl	Obsolete. Allocates and initializes an MDL describing a buffer specified by the given virtual address and length in bytes; returns a pointer to the MDL.
MmGetMdlByteCount	Returns the length in bytes of the buffer mapped by a given MDL.
MmGetMdlByteOffset	Returns the byte offset within a page of the buffer described by a given MDL.
MmGetMdlVirtualAddress	Returns a (possibly invalid) virtual address for a buffer described by a given MDL; the returned address, used as an index to a physical address entry in the MDL, can be input to MapTransfer for drivers that use DMA.
MmGetPhysicalAddress	Returns the corresponding physical address for a given valid virtual address.
MmGetSystemAddressForMdlSafe	Returns a system-space virtual address that maps the physical pages described by a given MDL for drivers whose devices must use programmed I/O (PIO). If no virtual address exists, one is assigned.
MmInitializeMdl	Initializes a caller-created MDL to describe a buffer specified by the given virtual address and length in bytes.
MmIsAddressValid	Returns whether a page fault will occur if a read or write operation is done at the given virtual address.
MmMapIoSpace	Maps a physical address range to a cached or noncached virtual address range in nonpaged system space.
MmMapLockedPages	Obsolete. Maps already locked physical pages, described by a given MDL, to a returned virtual address range.
MmMapLockedPagesWithReservedMapping	Maps a virtual address range already reserved with MmAllocateMappingAddress.
MmPrepareMdlForReuse	Reinitializes a caller-created MDL for reuse.

FUNCTION	DESCRIPTION
MmProbeAndLockPages	Probes the pages specified in an MDL for a particular kind of access, makes the pages resident, and locks them in memory; returns the MDL updated with corresponding physical addresses. (Usually, only highest-level drivers call this routine.)
MmProtectMdlSystemAddress	Sets the protection type for the memory-address range.
MmSecureVirtualMemory	Secures a memory-address range so that it cannot be freed and its page protection cannot be made more restrictive.
MmSizeOfMdl	Returns the number of bytes required for an MDL describing the buffer specified by the given virtual address and length in bytes.
MmUnlockPages	Unlocks the previously probed and locked pages specified in an MDL.
MmUnmapIoSpace	Unmaps a virtual address range from a physical address range.
MmUnmapLockedPages	Releases a mapping set up by MmMapLockedPages.
MmUnmapReservedMapping	Unmaps a virtual address range mapped by MmMapLockedPagesWithReservedMapping.
MmUnsecureVirtualMemory Unsecures a memory-address range secured by MmSecureVirtualMemory.	

Buffer and MDL Management

The buffer- and MDL-management macros are called by kernel-mode drivers to manage buffers and memory descriptor lists (MDLs).

For more information about MDLs, see Using MDLs.

FUNCTION	DESCRIPTION
ADDRESS_A	D_SIZE_TO_SPAN_PAGES
BYTE_OFFSET	Returns the byte offset of a given virtual address within the page.
BYTES_TO_PAGES	Returns the number of pages necessary to contain a given number of bytes.
PAGE_ALIGN	Returns the page-aligned virtual address for the page that contains a given virtual address.
ROUND_TO_PAGES	Rounds a given size in bytes up to a page-size multiple.

Device Memory Access

The device-memory-access macros are called by kernel-mode drivers to access the memory-mapped hardware registers and I/O ports of their respective devices.

For the following macros, XXX_REGISTER_XXX indicates device memory that is mapped onto system memory address space, while XXX_PORT_XXX indicates device memory in I/O port address space.

FUNCTION	DESCRIPTION
EAD_PORT_UCHAR	Reads a UCHAR value from the given I/O port address.
READ_PORT_USHORT	Reads a USHORT value from the given I/O port address.
READ_PORT_ULONG	Reads a ULONG value from the given I/O port address.
READ_PORT_BUFFER_UCHAR	Reads a given count of UCHAR values from the given I/O port into a given buffer.
READ_PORT_BUFFER_USHORT	Reads a given count of USHORT values from the given I/O port into a given buffer.
READ_PORT_BUFFER_ULONG	Reads a given count of ULONG values from the given I/O port into a given buffer.
WRITE_PORT_UCHAR	Writes a given UCHAR value to the given I/O port address.
WRITE_PORT_USHORT	Writes a given USHORT value to the given I/O port address.
WRITE_PORT_ULONG	Writes a given ULONG value to the given I/O port address.
WRITE_PORT_BUFFER_UCHAR	Writes a given count of UCHAR values from a given buffer to the given I/O port.
WRITE_PORT_BUFFER_USHORT	Writes a given count of USHORT values from a given buffer to the given I/O port.
WRITE_PORT_BUFFER_ULONG	Writes a given count of ULONG values from a given buffer to the given I/O port.
READ_REGISTER_UCHAR	Reads a UCHAR value from the given register address in memory space.
READ_REGISTER_USHORT	Reads a USHORT value from the given register address in memory space.
READ_REGISTER_ULONG	Reads a ULONG value from the given register address in memory space.
READ_REGISTER_BUFFER_UCHAR	Reads a given count of UCHAR values from the given register address into the given buffer.
READ_REGISTER_BUFFER_USHORT	Reads a given count of USHORT values from the given register address into the given buffer.
READ_REGISTER_BUFFER_ULONG	Reads a given count of ULONG values from the given register address into the given buffer.
WRITE_REGISTER_UCHAR	Writes a given UCHAR value to the given register address in memory space.

FUNCTION	DESCRIPTION
WRITE_REGISTER_USHORT	Writes a given USHORT value to the given register address in memory space.
WRITE_REGISTER_ULONG	Writes a given ULONG value to the given register address in memory space.
WRITE_REGISTER_BUFFER_UCHAR	Writes a given count of UCHAR values from a given buffer to the given register address.
WRITE_REGISTER_BUFFER_USHORT	Writes a given count of USHORT values from a given buffer to the given register address.
WRITE_REGISTER_BUFFER_ULONG	Writes a given count of ULONG values from a given buffer to the given register address.

Pageable Drivers

The pageable-driver routines are called by kernel-mode drivers to lock and unlock a driver's pageable code or data sections, or make an entire driver pageable.

FUNCTION	DESCRIPTION
M	LockPagableCodeSection
MmLockPagableDataSection	Locks data marked with a special compiler directive into system space, when that data is accessed infrequently, predictably, and at an IRQL of less than DISPATCH_LEVEL.
MmLockPagableSectionByHandle	Locks a pageable section into system memory using a handle returned from MmLockPagableCodeSection or MmLockPagableDataSection.
MmUnlockPagableImageSection	Releases a section that was previously locked into system space when the driver is no longer processing IRPs, or when the contents of the section are no longer required.
MmPageEntireDriver	Lets a driver page all of its code and data regardless of the attributes of the various sections in the driver's image.
MmResetDriverPaging	Resets a driver's pageable status to that specified by the sections making up the driver's image.

Sections and Views

The section- and view-management routines are called by kernel-mode drivers to set up mapped sections and views of memory.

FUNCTION	DESCRIPTION
nitializeObjectAttributes	Sets up a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a ZwCreateXxx or ZwOpenXxx routine.
ZwOpenSection	Obtains a handle for an existing section, provided that the requested access can be allowed.

FUNCTION	DESCRIPTION
ZwMapViewOfSection	Maps a view of an open section into the virtual address space of a process. Returns an offset into the section (base of the mapped view) and the size mapped.
ZwUnmapViewOfSection	Releases a mapped view in the virtual address space of a process.

Access to Structures

The structure-access macros are called by kernel-mode drivers to access parts of structures.

FUNCTION	DESCRIPTION
RGUMENT_PRESENT	Returns FALSE if an argument pointer is NULL; otherwise returns TRUE.
CONTAINING_RECORD	Returns the base address of an instance of a structure given the structure type and the address of a field within it.
FIELD_OFFSET	Returns the byte offset of a named field in a known structure type.

Plug and Play Routines

These routines are used by drivers to implement Plug and Play (PnP) support. For background and task-oriented information about supporting PnP in drivers, see Plug and Play.

The following topics summarize the routines by functionality:

Device Information Routines

FUNCTION	DESCRIPTION
oGetDeviceProperty	Retrieves information about a device such as configuration information and the name of its PDO.
IoInvalidateDeviceRelations	Notifies the PnP manager that the relations for a device have changed.
IoInvalidateDeviceState	Notifies the PnP manager that the PnP state of a device has changed. In response, the PnP manager sends an IRP_MN_QUERY_PNP_DEVICE_STATE to the device stack.
IoReportDetectedDevice	Reports a non-PnP device to the PnP manager.
IoReportResourceForDetection	Claims hardware resources in the configuration registry for a legacy device. This routine is for drivers that detect legacy hardware that cannot be enumerated by PnP.

Registry Routines

FUNCTION	DESCRIPTION

FUNCTION	DESCRIPTION
oOpenDeviceInterfaceRegistryKey	Returns a handle to a registry key for storing information about a particular device interface.
IoOpenDeviceRegistryKey	Returns a handle to a device-specific or a driver-specific registry key for a particular device instance.

Device Interface Routines

FUNCTION	DESCRIPTION
IoRegisterDeviceInterface	Registers device functionality (a device interface) that a driver will enable for use by applications or other system components.
IoSetDeviceInterfaceState	Enables or disables a previously registered device interface. Applications and other system components can open only interfaces that are enabled.
IoOpenDeviceInterfaceRegistryKey	Returns a handle to a registry key for storing information about a particular device interface.
IoGetDeviceInterfaces	Returns a list of device interfaces of a particular device interface class (such as all devices on the system that support a HID interface).
IoGetDeviceInterfaceAlias	Returns the alias device interface of the specified interface class, if the alias exists. Device interfaces are considered aliases if they are exposed by the same underlying device and have identical interface reference strings, but are of different interface classes.

PnP Notification Routines

FUNCTION	DESCRIPTION
IoRegisterPlugPlayNotification	Registers a driver callback routine to be called when the specified PnP event occurs.
IoReportTargetDeviceChange	Notifies the PnP manager that a custom event has occurred on a device. The PnP manager sends notification of the event to drivers that registered for it. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.
IoReportTargetDeviceChangeAsynchronous	Notifies the PnP manager that a custom event has occurred on a device. Returns immediately and does not wait while the PnP manager sends notification of the event to drivers that registered for it. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.
IoUnregisterPlugPlayNotification	Removes the registration of a driver's callback routine for a PnP event.

Remove Lock Routines

FUNCTION	DESCRIPTION
oInitializeRemoveLock	Initializes a remove lock for a device object. A driver can use the lock to track outstanding I/O on a device and to determine when the driver can delete its device object in response to an IRP_MN_REMOVE_DEVICE request.
IoAcquireRemoveLock	Increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack nor deleted.
IoReleaseRemoveLock	Releases a remove lock acquired with a previous call to IoAcquireRemoveLock.
IoReleaseRemoveLockAndWait	Releases a remove lock acquired with a previous call to IoAcquireRemoveLock and waits until all acquisitions of the lock have been released. A driver typically calls this routine in its dispatch code for an IRP_MN_REMOVE_DEVICE request.

Other PnP Routines

FUNCTION	DESCRIPTION
oAdjustPagingPathCount	Increments or decrements a caller-supplied page-file counter as an atomic operation. This routine can be used to adjust other counters, such as counters for hibernation files or crash-dump files.
IoRequestDeviceEject	Notifies the PnP manager that the device eject button was pressed. Note that this routine reports a request for a device eject, not media eject.

Windows Management Instrumentation Routines

This section summarizes kernel-mode support routines that drivers can use to interact with Windows Management Instrumentation (WMI).

The categories of support routines include those that drivers can call to:

- Process IRPs that a driver receives (WMI IRP Processing Routines).

FUNCTION	DESCRIPTION
WmiCompleteRequest	If a driver uses WmiSystemControl to dispatch the WMI IRP to a callback routine, then the callback routine can use WmiCompleteRequest to complete the IRP.
WmiSystemControl Dispatches a WMI IRP to a driver-supplied callback routine.	
WmiFireEvent	The WmiFireEvent routine sends an event to WMI for delivery to data consumers that have requested notification of the event.
WmiQueryTraceInformation	The WmiQueryTraceInformation routine returns information about a WMI event trace.

FUNCTION	DESCRIPTION
WmiSystemControl	The WmiSystemControl routine is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.
WmiTraceMessage	The WmiTraceMessage routine adds a message to the output log of a WPP software tracing session.
WmiTraceMessageVa	The WmiTraceMessageVa routine adds a message to the output log of a WPP software tracing session.

- Drivers can use these routines in processing WMI IRPs. Drivers use these routines to send WMI IRPs.

FUNCTION	DESCRIPTION
IoWMIAllocateInstanceIds	Allocates unused WMI instance IDs for a given WMI class GUID.
IoWMIDeviceObjectToInstanceName	Given a device object, determines the WMI class instance name supported by the corresponding driver. Callers can use this to determine the instance names supported by a particular driver.
IoWMIExecuteMethod	Executes the specified WMI class method.
IoWMIHandleToInstanceName	Given a file handle, determines the WMI class instance name supported by the corresponding driver. Callers can use this to determine the instance names supported by a particular driver.
IoWMIOpenBlock	Opens a WMI data block. Callers use this to submit WMI I/O requests.
IoWMIQueryAllData	Fetches the property values for every instance of the specified WMI class GUID.
IoWMIQueryAllDataMultiple	Fetches the property values for every instance of the specified set of WMI class GUIDs.
IoWMIQuerySingleInstance	Fetches the property values for a particular instance of the specified WMI class GUID.
IoWMIQuerySingleInstanceMultiple	Fetches the property values for a particular set of WMI class instances.
IoWMISetNotificationCallback	Sets a notification callback for WMI events.
IoWMISetSingleInstance	Sets the property values for a particular WMI class instance.
IoWMISetSingleItem	Sets the specified property for a particular WMI class instance.

This section describes required and optional routines that a kernel-mode WDM driver contains, if the driver handles WMI Minor IRPs by calling WmiSystemControl. For more information, see Calling WmiSystemControl to

Handle WMI IRPs.

The DpWmiXxx names that are used in the Microsoft Windows Driver Kit (WDK) documentation are placeholders. A driver's DpWmiXxx routines can have any names that the driver writer chooses.

ZwXxx / NtXxx Routines

The ZwXxx routines provide a set of system entry points that parallel some of the executive's system services. Calling a ZwXxx routine from kernel-mode code results in a call to the corresponding system service. Calling a ZwXxx routine from user mode is not supported; instead, native applications (applications that bypass the Microsoft Win32 subsystem) should call the NtXxx equivalent of the ZwXxx routine.

For a list of NtXxx routines, see NtXxx Routines.

For a call to a ZwXxx routine from a kernel-mode driver, the system does not check the caller's access rights, nor does it set the previous processor mode to UserMode. Before calling a ZwXxx routine, a kernel-mode driver must check all user-supplied parameters for validity.

For more information about the relationship between NtXxx and ZwXxx routines, see Using Nt and Zw Versions of the Native System Services Routines. For a list of the ZwXxx routines in each major functional category, see Summary of Kernel-Mode Support Routines.

The following routines are reserved for system use. Do not use them in your driver.

|Routine|Replacement| |ZwCancelTimer| Use KeCancelTimer instead.| |ZwCreateTimer| Use KeInitializeTimer or KeInitializeTimerEx instead.| |ZwOpenTimer.|| |ZwSetTimer| Use KeSetTimer instead.| |NtRenameTransactionManager| Obsolete.|

> **NOTE**
> NtRenameTransactionManager and TmRenameTransactionManager are two versions of the same routine. Kernel-mode drivers should not call NtRenameTransactionManager. They should instead call TmRenameTransactionManager.

FUNCTION	DESCRIPTION
NtRenameTransactionManager	The NtRenameTransactionManager routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.
NtSetInformationTransactionManager Do not call this routine from kernel-mode code.	
ZwAllocateLocallyUniqueId	The ZwAllocateLocallyUniqueId routine allocates a locally unique identifier (LUID).
ZwAllocateVirtualMemory	The ZwAllocateVirtualMemory routine reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process.
ZwClose	The ZwClose routine closes an object handle.
ZwCreateDirectoryObject	The ZwCreateDirectoryObject routine creates or opens an object-directory object.

FUNCTION	DESCRIPTION
ZwCreateEvent	The ZwCreateEvent routine creates an event object, sets the initial state of the event to the specified value, and opens a handle to the object with the specified desired access.
ZwCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
ZwCreateKey	The ZwCreateKey routine creates a new registry key or opens an existing one.
ZwCreateKeyTransacted	The ZwCreateKeyTransacted routine creates a new registry key or opens an existing one, and it associates the key with a transaction.
ZwCreateSection	The ZwCreateSection routine creates a section object.
ZwDeleteFile	The ZwDeleteFile routine deletes the specified file.
ZwDeleteKey	The ZwDeleteKey routine deletes an open key from the registry.
ZwDeleteValueKey	The ZwDeleteValueKey routine deletes a value entry matching a name from an open key in the registry. If no such entry exists, an error is returned.
ZwDeviceIoControlFile	The ZwDeviceIoControlFile routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.
ZwDuplicateToken	The ZwDuplicateToken function creates a handle to a new access token that duplicates an existing token. This function can create either a primary token or an impersonation token.
ZwEnumerateKey	The ZwEnumerateKey routine returns information about a subkey of an open registry key.
ZwEnumerateValueKey	The ZwEnumerateValueKey routine gets information about the value entries of an open key.
ZwFlushBuffersFile	The ZwFlushBuffersFile routine is called by a file system filter driver to send a flush request for the specified file to the file system.
ZwFlushBuffersFileEx	The ZwFlushBuffersFileEx routine is called by a file system filter driver to send a flush request for a given file to the file system. An optional flush operation flag can be set to control how file data is written to storage.
ZwFlushKey	The ZwFlushKey routine forces a registry key to be committed to disk.
ZwFlushVirtualMemory	The ZwFlushVirtualMemory routine flushes a range of virtual addresses within the virtual address space of a specified process which map to a data file back out to the data file if they have been modified.

FUNCTION	DESCRIPTION
ZwFreeVirtualMemory	The ZwFreeVirtualMemory routine releases, decommits, or both, a region of pages within the virtual address space of a specified process.
ZwFsControlFile	The ZwFsControlFile routine sends a control code directly to a specified file system or file system filter driver, causing the corresponding driver to perform the specified action.
ZwLoadDriver	The ZwLoadDriver routine loads a driver into the system.
ZwLockFile	The ZwLockFile routine requests a byte-range lock for the specified file.
ZwMakeTemporaryObject	The ZwMakeTemporaryObject routine changes the attributes of an object to make it temporary.
ZwMapViewOfSection	The ZwMapViewOfSection routine maps a view of a section into the virtual address space of a subject process.
ZwNotifyChangeKey	The ZwNotifyChangeKey routine allows a driver to request notification when a registry key changes.
ZwOpenEvent	The ZwOpenEvent routine opens a handle to an existing named event object with the specified desired access.
ZwOpenFile	The ZwOpenFile routine opens an existing file, directory, device, or volume.
ZwOpenKey	The ZwOpenKey routine opens an existing registry key.
ZwOpenKeyEx	The ZwOpenKeyEx routine opens an existing registry key.
ZwOpenKeyTransacted	The ZwOpenKeyTransacted routine opens an existing registry key and associates the key with a transaction.
ZwOpenKeyTransactedEx	The ZwOpenKeyTransactedEx routine opens an existing registry key and associates the key with a transaction.
ZwOpenProcess	The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.
ZwOpenProcessTokenEx	The ZwOpenProcessTokenEx routine opens the access token associated with a process.
ZwOpenSection	The ZwOpenSection routine opens a handle for an existing section object.
ZwOpenSymbolicLinkObject	The ZwOpenSymbolicLinkObject routine opens an existing symbolic link.

FUNCTION	DESCRIPTION
ZwOpenThreadTokenEx	The ZwOpenThreadTokenEx routine opens the access token associated with a thread.
ZwPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.
ZwQueryInformationThread	The ZwQueryInformationThread routine retrieves information about the specified thread, such as its page priority.
ZwQueryDirectoryFile	The ZwQueryDirectoryFile routine returns various kinds of information about files in the directory specified by a given file handle.
ZwQueryEaFile	The ZwQueryEaFile routine returns information about extended-attribute (EA) values for a file.
ZwQueryFullAttributesFile	The ZwQueryFullAttributesFile routine supplies network open information for the specified file.
ZwQueryInformationFile	The ZwQueryInformationFile routine returns various kinds of information about a file object.
ZwQueryInformationToken	The ZwQueryInformationToken routine retrieves a specified type of information about an access token. The calling process must have appropriate access rights to obtain the information.
ZwQueryKey	The ZwQueryKey routine provides information about the class of a registry key, and the number and sizes of its subkeys.
ZwQueryObjec	The ZwQueryObject routine provides information about a supplied object.
ZwQueryQuotaInformationFile	The ZwQueryQuotaInformationFile routine retrieves quota entries associated with the volume specified by the FileHandle parameter.
ZwQuerySecurityObject	The ZwQuerySecurityObject routine retrieves a copy of an object's security descriptor.
ZwQuerySymbolicLinkObject	The ZwQuerySymbolicLinkObject routine returns a Unicode string that contains the target of a symbolic link.
ZwQueryValueKey	The ZwQueryValueKey routine returns a value entry for a registry key.
ZwQueryVirtualMemory	The ZwQueryVirtualMemory routine determines the state, protection, and type of a region of pages within the virtual address space of the subject process.
ZwQueryVolumeInformationFile	The ZwQueryVolumeInformationFile routine retrieves information about the volume associated with a given file, directory, storage device, or volume.
ZwReadFile	The ZwReadFile routine reads data from an open file.

FUNCTION	DESCRIPTION
ZwSetEaFile	The ZwSetEaFile routine sets extended-attribute (EA) values for a file.
ZwSetEvent	The ZwSetEvent routine sets an event object to a Signaled state and attempts to satisfy as many waits as possible.
ZwSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
ZwSetInformationThread	The ZwSetInformationThread routine sets the priority of a thread.
ZwSetInformationToken The ZwSetInformationToken routine modifies information in a specified token. The calling process must have appropriate access rights to set the information.	
ZwSetInformationVirtualMemory	The ZwSetInformationVirtualMemory routine performs an operation on a specified list of address ranges in the user address space of a process.
ZwSetQuotaInformationFile	The ZwSetQuotaInformationFile routine changes quota entries for the volume associated with the FileHandle parameter. All of the quota entries in the specified buffer are applied to the volume.
ZwSetSecurityObject	The ZwSetSecurityObject routine sets an object's security state.
ZwSetValueKey	The ZwSetValueKey routine creates or replaces a registry key's value entry.
ZwSetVolumeInformationFile	The ZwSetVolumeInformationFile routine modifies information about the volume associated with a given file, directory, storage device, or volume.
ZwTerminateProcess	The ZwTerminateProcess routine terminates a process and all of its threads.
ZwUnloadDriver	The ZwUnloadDriver routine unloads a driver from the system. Use this routine with extreme caution. (See the following Remarks section.)
ZwUnlockFile	The ZwUnlockFile routine unlocks a byte-range lock in a file.
ZwUnmapViewOfSection	The ZwUnmapViewOfSection routine unmaps a view of a section from the virtual address space of a subject process.
ZwWaitForSingleObject	The ZwWaitForSingleObject routine waits until the specified object attains a state of Signaled. An optional time-out can also be specified.
ZwWriteFile	The ZwWriteFile routine writes data to an open file.

Auxiliary Kernel-Mode Library Routines and Structures

The Auxiliary Kernel-Mode Library enables drivers to access some system capabilities that are not available from kernel-mode subsystems.

The AuxKlibInitialize routine initializes the Auxiliary Kernel-Mode Library. Drivers that use this library must call AuxKlibInitialize before calling any of the library's other routines.

- AuxKlibEnumerateSystemFirmwareTables
- AuxKlibGetBugCheckData
- AuxKlibGetImageExportDirectory
- AuxKlibGetSystemFirmwareTable
- AuxKlibInitialize
- AuxKlibQueryModuleInformation
- AUX_MODULE_BASIC_INFO
- AUX_MODULE_EXTENDED_INFO
- KBUGCHECK_DATA

Processor Group Compatibility Library

Support is available for kernel-mode drivers that use processor groups. The Processor Group (ProcGrp) compatibility library enables a kernel-mode driver that is written to use processor groups run on earlier versions of Windows, which do not support processor groups. When this driver runs on, it can take advantage of multiple processor groups, if the hardware platform supports them. When this same driver runs on an earlier version of Windows, it is limited to a single processor group, regardless of the capabilities of the hardware platform. To use the ProcGrp library, the driver must call a library initialization routine. Additionally, the driver must be compiled for the Windows 7 environment in the WDK, and must be linked with Procgrp.lib.

The ProcGrp library is designed to meet the compatibility requirements of drivers that call the following KeXxx routines but must also run on versions of Windows that do not implement these routines:

- KeGetCurrentProcessorNumberEx
- KeGetProcessorIndexFromNumber
- KeGetProcessorNumberFromIndex
- KeQueryActiveGroupCount
- KeQueryActiveProcessorCountEx
- KeQueryGroupAffinity
- KeQueryMaximumProcessorCount
- KeQueryMaximumProcessorCountEx
- KeQueryMaximumGroupCount
- KeSetSystemAffinityThreadEx
- KeSetSystemGroupAffinityThread
- KeRevertToUserAffinityThreadEx
- KeRevertToUserGroupAffinityThread
- KeSetTargetProcessorDpcEx

The ProcGrp library implements wrapper functions for the KeXxx routines in the preceding list. If the library runs on Windows 7 or Windows Server 2008 R2, the wrapper functions simply call the corresponding KeXxx routines. The wrapper functions have the same names as these KeXxx routines, and their behavior is identical to that of the KeXxx routines that they replace.

Earlier versions of Windows do not support processor groups and do not implement the KeXxx routines in the preceding list. If the ProcGrp library is linked to a driver that runs on one of these earlier versions of Windows, the library initialization function, WdmlibProcgrpInitialize, detects that the operating system does not support

processor groups. To deal with this case, each wrapper function contains a simplified implementation of the corresponding KeXxx routine. This implementation supports only one processor group, group number 0. For example, the wrapper function for the KeQueryMaximumGroupCount routine always returns a count of one. For another example, the wrapper function for the KeGetCurrentProcessorNumberEx routine emulates this routine by calling the KeGetCurrentProcessorNumber routine. KeGetCurrentProcessorNumber is similar to KeGetCurrentProcessorNumberEx, but lacks support for processor groups, which, in this case, has the same effect as supporting just one processor group.

For more information about the support for processor groups in Windows 7, see the Supporting Systems That Have More Than 64 Processors white paper on the WHDC website.

The ProcGrp library is contained in the Windows 7 version of the WDK. The library functions are declared in the Procgrp.h header file and are implemented in the Procgrp.lib library file.

In addition to the KeXxx wrapper functions, the ProcGrp library implements the following function to initialize the library:

- WdmlibProcgrpInitialize

DMA

DMA Library Routines Drivers use the routines documented in this section to perform direct memory access (DMA) operations. The routines are accessed through pointers and cannot be called directly by name.

Drivers performing DMA operations use IoGetDmaAdapter to get a pointer to the DMA_ADAPTER structure for the device. The DmaOperations member of the structure points to a DMA_OPERATIONS structure, which is a table of pointers to the DMA routines for the physical device object of that device.

FUNCTION	DESCRIPTION
IoGetDmaAdapter	Returns a pointer to an adapter object that represents either the DMA channel to which the driver's device is connected or the driver's bus-master adapter. Also returns the maximum number of map registers the driver can specify for each DMA transfer.
MmGetMdlVirtualAddress	Returns the base virtual address of a buffer described by a given MDL. The returned address, used as an index to a physical address entry in the MDL, can be input to MapTransfer.
MmGetSystemAddressForMdlSafe	Returns a nonpaged system-space virtual address for the base of the memory area described by an MDL. It maps the physical pages described by the MDL into system space, if they are not already mapped to system space.
ADDRESS_AND_SIZE_TO_SPAN_PAGES	Returns the number of pages spanned by the virtual range defined by a virtual address and a length in bytes. A driver can use this macro to determine whether a transfer request must be split into partial transfers.
AllocateAdapterChannel	Reserves exclusive access to a DMA channel and map registers for a device. When the channel and registers are available, this routine calls a driver-supplied AdapterControl routine to carry out an I/O operation through either the system DMA controller or a bus-master adapter.

FUNCTION	DESCRIPTION
AllocateCommonBuffer	Allocates and maps a logically contiguous region of memory that is simultaneously accessible from both the processor and a device. This routine returns TRUE if the requested length was allocated.
BuildMdlFromScatterGatherList	Builds an MDL corresponding to a scatter/gather list.
BuildScatterGatherList	Prepares the system for scatter/gather DMA for a device and calls a driver-supplied routine to carry out the I/O operation. This function provides the same functionality as GetScatterGatherList, except it uses a driver-supplied buffer to hold the scatter/gather list.
CalculateScatterGatherList	Computes the buffer size needed to hold a scatter/gather list for a memory buffer.
FlushAdapterBuffers	Forces any data remaining in either a bus-master adapter's or the system DMA controller's internal buffers to be written into memory or to the device.
FreeAdapterChannel	Releases an adapter object that represents a system DMA channel, and optionally releases any allocated map registers.
FreeCommonBuffer	Releases and unmaps a previously allocated common buffer. Arguments must match those passed in an earlier call to AllocateCommonBuffer.
FreeMapRegisters	Releases a set of map registers that were saved from a call to AllocateAdapterChannel. A driver calls this routine after using the registers in one or more calls to MapTransfer, flushing the cache by calling FlushAdapterBuffers, and completing the bus-master DMA transfer.
GetDmaAlignment	Returns the buffer alignment requirements for a DMA controller or device.
GetScatterGatherList	Prepares the system for scatter/gather DMA for a device and calls a driver-supplied routine to carry out the I/O operation. For devices that support scatter/gather DMA, this routine combines the functionality of AllocateAdapterChannel and MapTransfer.
KeFlushIoBuffers	Flushes the memory region described by an MDL from all processors' caches into memory.
MapTransfer	Sets up map registers for an adapter object previously allocated by AllocateAdapterChannel to map a transfer from a locked-down buffer. Returns the logical address of the mapped region and, for bus-master devices that support scatter/gather, the number of bytes mapped.
PutDmaAdapter	Frees an adapter object previously allocated by IoGetDmaAdapter.

FUNCTION	DESCRIPTION
PutScatterGatherList	Frees map registers and scatter/gather list previously allocated by GetScatterGatherList.
ReadDmaCounter Returns the number of bytes yet to be transferred during the current system DMA operation (in autoinitialize mode).	

PIO

FUNCTION	DESCRIPTION
MmProbeAndLockPages	Probes the pages specified in an MDL for a particular kind of access, makes the pages resident, and locks them in memory; returns the MDL updated with corresponding physical addresses.
MmGetSystemAddressForMdlSafe	Returns a system-space virtual address that maps the physical pages described by a given MDL for drivers whose devices must use PIO. If no virtual address exists, one is assigned.
KeFlushIoBuffers	Flushes the memory region described by a given MDL from all processors' caches into memory.
MmUnlockPages	Unlocks the previously probed and locked pages specified in an MDL.
MmMapIoSpace	Maps a physical address range to a cached or noncached virtual address range in nonpaged system space.
MmUnmapIoSpace	Unmaps a virtual address range from a physical address range.

Interrupts

FUNCTION	DESCRIPTION
IoConnectInterrupt	Registers a driver's interrupt handling routine. Drivers should use IoConnectInterruptEx instead.
IoDisconnectInterrupt	Unregisters an interrupt handling routine that IoConnectInterrupt registered.
IoConnectInterruptEx	Registers a driver's interrupt handling routine. Drivers can register either an InterruptService routine for line-based interrupts or an InterruptMessageService routine for message-signaled interrupts.
IoDisconnectInterruptEx	Unregisters an interrupt-handling routine that IoConnectInterruptEx registered.

FUNCTION	DESCRIPTION
IoInitializeDpcRequest	Associates a driver-supplied DpcForIsr routine with a given device object, so that the DpcForIsr routine can complete interrupt-driven I/O operations.
KeSynchronizeExecution	Synchronizes the execution of a driver-supplied SynchCritSection routine with that of the ISR associated with a set of interrupt objects, given a pointer to the interrupt objects.
KeAcquireInterruptSpinLock	Acquires the spin lock that synchronizes access with an interrupt's ISR.
KeReleaseInterruptSpinLock	Releases the spin lock that synchronized access with an interrupt's ISR.
KeRegisterNmiCallback	Registers a routine to be called whenever a nonmaskable interrupt (NMI) occurs.
KeDeregisterNmiCallback	Deregisters a routine registered by KeRegisterNmiCallback.

Driver-Managed Queues

FUNCTION	DESCRIPTION
KeInitializeSpinLock	Initializes a variable of type KSPIN_LOCK. An initialized spin lock is a required parameter to the ExInterlockedXxxList routines.
InitializeListHead	Sets up a queue header for a driver's internal queue, given a pointer to driver-supplied storage for the queue header and queue. An initialized queue header is a required parameter to the ExInterlockedInsert/RemoveXxxList routines.
ExInterlockedInsertTailList	Inserts an entry at the tail of a doubly linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the list links.
ExInterlockedInsertHeadList	Inserts an entry at the head of a doubly linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the links in the list.
ExInterlockedRemoveHeadList	Removes an entry from the head of a doubly linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the links in the list.
ExInterlockedPopEntryList	Removes an entry from the head of a singly linked list as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the list.
ExInterlockedPushEntryList	Inserts an entry at the head of a singly linked list as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the list.

FUNCTION	DESCRIPTION
IsListEmpty	Returns TRUE if a queue is empty. (This type of doubly linked list is not protected by a spin lock, unless the caller explicitly manages synchronization to queued entries with an initialized spin lock for which the caller supplies the storage.)
InsertTailList	Queues an entry at the end of the list.
InsertHeadList	Queues an entry at the head of the list.
RemoveHeadList	Dequeues an entry at the head of the list.
RemoveTailList	Dequeues an entry at the end of the list.
RemoveEntryList	Returns whether a given entry is in the given list and dequeues the entry if it is.
PushEntryList	Inserts an entry into the queue. (This type of singly linked list is not protected by a spin lock, unless the caller explicitly manages synchronization to queued entries with an initialized spin lock for which the caller supplies the storage.)
PopEntryList	Removes an entry from the queue.
ExInterlockedPopEntrySList	Removes an entry from the head of a sequenced, singly linked list that was set up with ExInitializeSListHead.
ExInterlockedPushEntrySList	Queues an entry at the head of a sequenced, singly linked list that was set up with ExInitializeSListHead.
ExQueryDepthSList	Returns the number of entries currently queued in a sequenced, singly linked list.
ExInitializeNPagedLookasideList	Sets up a lookaside list, protected by a system-supplied spin lock, in nonpaged pool from which the driver can allocate and free blocks of a fixed size.
KeInitializeDeviceQueue	Initializes a device queue object to a not-busy state, setting up an associated spin lock for multiprocessor-safe access to device queue entries.
KeInsertDeviceQueue	Acquires the device queue spin lock and queues an entry to a device driver if the device queue is not empty; otherwise, inserts the entry at the tail of the device queue.
KeInsertByKeyDeviceQueue	Acquires the device queue spin lock and queues an entry to a device driver if the device queue is not empty; otherwise, inserts the entry into the queue according to the given sort-key value.
KeRemoveDeviceQueue	Removes an entry from the head of a given device queue.
KeRemoveByKeyDeviceQueue	Removes an entry, selected according to the specified sort-key value, from the given device queue.

FUNCTION	DESCRIPTION
KeRemoveEntryDeviceQueue	Determines whether a given entry is in the given device queue and, if so, dequeues the entry.

Driver System Processes and Threads

FUNCTION	DESCRIPTION
PsCreateSystemThread	Creates a kernel-mode thread associated with a given process object or with the default system process. Returns a handle for the thread.
PsTerminateSystemThread	Terminates the current thread and satisfies as many waits as possible for the current thread object.
PsGetCurrentThread	Returns a handle for the current thread.
KeGetCurrentThread	Returns a pointer to the opaque thread object that represents the current thread.
KeQueryPriorityThread	Returns the current priority of a given thread.
KeSetBasePriorityThread	Sets up the run-time priority, relative to the system process, for a driver-created thread.
KeSetPriorityThread	Sets up the run-time priority for a driver-created thread with a real-time priority attribute.
KeDelayExecutionThread	Puts the current thread into an alertable or nonalertable wait state for a given interval.
IoQueueWorkItem	Queues an initialized work queue item so the driver-supplied routine will be called when a system worker thread is given control.
ZwSetInformationThread	Sets the priority of a given thread for which the caller has a handle.

FUNCTION	DESCRIPTION
PsGetCurrentProcessId	The PsGetCurrentProcessId routine identifies the current thread's process.
PsGetProcessCreateTimeQuadPart	The PsGetProcessCreateTimeQuadPart routine returns a LONGLONG value that represents the time at which the process was created.
PsGetProcessId	The PsGetProcessId routine returns the process identifier (process ID) that is associated with a specified process.
PsQueryTotalCycleTimeProcess	The PsQueryTotalCycleTimeProcess routine returns the accumulated cycle time for the specified process.

FUNCTION	DESCRIPTION
PCREATE_PROCESS_NOTIFY_ROUTINE	Process-creation callback implemented by a driver to track the system-wide creation and deletion of processes against the driver's internal state.
PsSetCreateProcessNotifyRoutine	The PsSetCreateProcessNotifyRoutine routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.
PCREATE_PROCESS_NOTIFY_ROUTINE_EX	A callback routine implemented by a driver to notify the caller when a process is created or exits.
PsSetCreateProcessNotifyRoutineEx	The PsSetCreateProcessNotifyRoutineEx routine registers or removes a callback routine that notifies the caller when a process is created or exits.
PsSetCreateProcessNotifyRoutineEx2	The PsSetCreateProcessNotifyRoutineEx2 routine registers or removes a callback routine that notifies the caller when a process is created or deleted.

FUNCTION	DESCRIPTION
PLOAD_IMAGE_NOTIFY_ROUTINE	Called by the operating system to notify the driver when a driver image or a user image (for example, a DLL or EXE) is mapped into virtual memory.
PsSetLoadImageNotifyRoutine	The PsSetLoadImageNotifyRoutine routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
PsSetLoadImageNotifyRoutineEx	The PsSetLoadImageNotifyRoutineEx routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
PsTerminateSystemThread	The PsTerminateSystemThread routine terminates the current system thread.

Best practices for implementing process and thread-related callback functions

This set of guidelines applies to these callback routines:

- PCREATE_PROCESS_NOTIFY_ROUTINE
- PCREATE_PROCESS_NOTIFY_ROUTINE_EX
- PCREATE_THREAD_NOTIFY_ROUTINE
- PLOAD_IMAGE_NOTIFY_ROUTINE

Keep notify routines short and simple.

- Do not make calls into a user mode service to validate the process, thread, or image.
- Do not make registry calls.
- Do not make blocking and/or Interprocess Communication (IPC) function calls.
- Do not synchronize with other threads because it can lead to reentrancy deadlocks.
- Use System Worker Threads to queue work especially work involving:
    - Slow API's or API's that call into other process.

- Any blocking behavior which could interrupt threads in core services.

- Be considerate of best practices for kernel mode stack usage. For examples, see How do I keep my driver from running out of kernel-mode stack? and Key Driver Concepts and Tips.

Run-Time Library (RTL) Routines

For information about functions that copy, concatenate, and format strings in a manner that prevents buffer overrun errors, see Safe String Functions, below. Other string manipulation functions include the following:

FUNCTION	DESCRIPTION
RtlInitString	Initializes the specified string in a buffer.
RtlInitAnsiString	Initializes the specified ANSI string in a buffer.
RtlInitUnicodeString	Initializes the specified Unicode string in a buffer.
RtlAnsiStringToUnicodeSize	Returns the size in bytes required to hold a Unicode version of a given buffered ANSI string.
RtlAnsiStringToUnicodeString	Converts a buffered ANSI string to a Unicode string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.) You can also use the string manipulation routines provided by a compiler to convert ANSI strings to Unicode.
RtlFreeUnicodeString	Releases a buffer containing a Unicode string, given a pointer to the buffer returned by RtlAnsiStringToUnicodeString.
RtlUnicodeStringToAnsiString	Converts a buffered Unicode string to an ANSI string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)
RtlFreeAnsiString	Releases a buffer containing an ANSI string, given a pointer to the buffer returned by RtlUnicodeStringToAnsiString.
RtlAppendUnicodeStringToString	Concatenates a copy of a buffered Unicode string with a buffered Unicode string, given pointers to both buffers.
RtlAppendUnicodeToString	Concatenates a given input string with a buffered Unicode string, given a pointer to the buffer.
RtlCopyString	Copies the source string to the destination, given pointers to both buffers, or sets the length of the destination string (but not the length of the destination buffer) to zero if the optional pointer to the source-string buffer is NULL.
RtlCopyUnicodeString	Copies the source string to the destination, given pointers to both buffers, or sets the length of the destination string (but not the length of the destination buffer) to zero if the optional pointer to the source-string buffer is NULL.

FUNCTION	DESCRIPTION
RtlEqualString	Returns TRUE if the given ANSI alphabetic strings are equivalent.
RtlEqualUnicodeString	Returns TRUE if the given buffered strings are equivalent.
RtlCompareString	Compares two buffered, single-byte character strings and returns a signed value indicating whether they are equivalent or which is greater.
RtlCompareUnicodeString	Compares two buffered Unicode strings and returns a signed value indicating whether they are equivalent or which is greater.
RtlUpperString	Converts a copy of a buffered string to uppercase and stores the copy in a destination buffer.
RtlUpcaseUnicodeString	Converts a copy of a buffered Unicode string to uppercase and stores the copy in a destination buffer.
RtlIntegerToUnicodeString	Converts an unsigned integer value in the specified base to one or more Unicode characters in a buffer.
RtlUnicodeStringToInteger	RtlUnicodeStringToInteger converts the Unicode string representation of an integer into its integer equivalent.

The following routines are reserved for system use. Do not use them in your driver.

ROUTINE	REPLACEMENT
RtlAssert	Use ASSERT instead.
RtlGetCallersAddress	Use the intrinsic _ReturnAddress instead.
RtlInterlockedAndBits	Use InterlockedAnd instead.
RtlInterlockedAndBitsDiscardReturn	Use InterlockedAnd instead.
RtlInterlockedClearBits	Use InterlockedAnd instead.
RtlInterlockedClearBitsDiscardReturn	Use InterlockedAnd instead.
RtlInterlockedSetBits	Use InterlockedOr instead.
RtlInterlockedSetBitsDiscardReturn	Use InterlockedOr instead.
RtlInterlockedSetClearBits	
RtlInterlockedXorBits	Use InterlockedXor instead
RtlWalkFrameChain	

Safe String Functions for Unicode and ANSI Characters

Use the functions in this section to manipulate Unicode and ANSI strings in kernel-mode drivers.

Each function is available in two versions:

A W-suffixed version that supports two-byte Unicode characters.

An A-suffixed version that supports one-byte ANSI characters.

If you use the safe string functions instead of the string manipulation functions that are provided by C-language run-time libraries, you protect your code from buffer overrun errors that can make code untrustworthy. For more information, see Using Safe String Functions.

FUNCTION	DESCRIPTION
RtlStringCbCatW	The RtlStringCbCatW and RtlStringCbCatA functions concatenate two byte-counted strings.
RtlStringCbCatExW	The RtlStringCbCatExW and RtlStringCbCatExA functions concatenate two byte-counted strings.
RtlStringCbCatNW	The RtlStringCbCatNW and RtlStringCbCatNA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNExW	The RtlStringCbCatNExW and RtlStringCbCatNExA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCopyW	The RtlStringCbCopyW and RtlStringCbCopyA functions copy a byte-counted string into a buffer.
RtlStringCbCopyExW	The RtlStringCbCopyExW and RtlStringCbCopyExA functions copy a byte-counted string into a buffer.
RtlStringCbCopyNW	The RtlStringCbCopyNW and RtlStringCbCopyNA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNExW	The RtlStringCbCopyNExW and RtlStringCbCopyNExA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbLengthW	The RtlStringCbLengthW and RtlStringCbLengthA functions determine the length, in bytes, of a supplied string.
RtlStringCbPrintfW	The RtlStringCbPrintfW and RtlStringCbPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfExW	The RtlStringCbPrintfExW and RtlStringCbPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfW	The RtlStringCbVPrintfW and RtlStringCbVPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.

FUNCTION	DESCRIPTION
RtlStringCbVPrintfExW	The RtlStringCbVPrintfExW and RtlStringCbVPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchCatW	The RtlStringCchCatW and RtlStringCchCatA functions concatenate two character-counted strings.
RtlStringCchCatExW	The RtlStringCchCatExW and RtlStringCchCatExA functions concatenate two character-counted strings.
RtlStringCchCatNW	The RtlStringCchCatNW and RtlStringCchCatNA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNExW	The RtlStringCchCatNExW and RtlStringCchCatNExA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCopyW	The RtlStringCchCopyW and RtlStringCchCopyA functions copy a null-terminated source string into a destination buffer of specified length.
RtlStringCchCopyExW	The RtlStringCchCopyExW and RtlStringCchCopyExA functions copy a character-counted string into a buffer.
RtlStringCchCopyNW	The RtlStringCchCopyNW and RtlStringCchCopyNA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNExW	The RtlStringCchCopyNExW and RtlStringCchCopyNExA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchLengthW	The RtlStringCchLengthW and RtlStringCchLengthA functions determine the length, in characters, of a supplied string.
RtlStringCchPrintfW	The RtlStringCchPrintfW and RtlStringCchPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfExW	The RtlStringCchPrintfExW and RtlStringCchPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfW	The RtlStringCchVPrintfW and RtlStringCchVPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfExW	The RtlStringCchVPrintfExW and RtlStringCchVPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlUnalignedStringCbLength	The RtlUnalignedStringCbLengthW function is a version of the RtlStringCbLength function that accepts an unaligned pointer to a string of Unicode characters.

FUNCTION	DESCRIPTION
RtlUnalignedStringCchLengthW	The RtlUnalignedStringCchLengthW function is a version of the RtlStringCchLength function that accepts an unaligned pointer to a string of Unicode characters.

Safe String Functions for UNICODE_STRING Structures

Use the functions in this section to manipulate strings within UNICODE_STRING structures in kernel-mode drivers.

If you use the safe string functions instead of the string manipulation functions that C-language run-time libraries provide, you protect your code from buffer overrun errors that can make code untrustworthy. For more information about safe string functions, see Using Safe String Functions.

FUNCTION	DESCRIPTION
RtlStringCbCopyUnicodeString	The RtlStringCbCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCbCopyUnicodeStringEx	The RtlStringCbCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyUnicodeString	The RtlStringCchCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyUnicodeStringEx	The RtlStringCchCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlUnicodeStringCat	The RtlUnicodeStringCat function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatEx	The RtlUnicodeStringCatEx function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatString	The RtlUnicodeStringCatString function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCatStringEx	The RtlUnicodeStringCatStringEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCbCatN	The RtlUnicodeStringCbCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCbCatNEx	The RtlUnicodeStringCbCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.

FUNCTION	DESCRIPTION
RtlUnicodeStringCbCatStringN	The RtlUnicodeStringCbCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCbCatStringNEx	The RtlUnicodeStringCbCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCbCopyN	The RtlUnicodeStringCbCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCbCopyNEx	The RtlUnicodeStringCbCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringN	The RtlUnicodeStringCbCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringNEx	The RtlUnicodeStringCbCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCchCatN	The RtlUnicodeStringCchCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatNEx	The RtlUnicodeStringCchCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatStringN	The RtlUnicodeStringCchCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCatStringNEx	The RtlUnicodeStringCchCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCopyN	The RtlUnicodeStringCchCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyNEx	The RtlUnicodeStringCchCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyStringN	The RtlUnicodeStringCchCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

FUNCTION	DESCRIPTION
RtlUnicodeStringCchCopyStringNEx	The RtlUnicodeStringCchCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCopy	The RtlUnicodeStringCopy function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyEx	The RtlUnicodeStringCopyEx function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyString	The RtlUnicodeStringCopyString function copies a string into a UNICODE_STRING structure.
RtlUnicodeStringCopyStringEx	The RtlUnicodeStringCopyStringEx function copies a string into a UNICODE_STRING structure.
RtlUnicodeStringInit	The RtlUnicodeStringInit function initializes a UNICODE_STRING structure.
RtlUnicodeStringInitEx	The RtlUnicodeStringInitEx function initializes a UNICODE_STRING structure.
RtlUnicodeStringPrintf	The RtlUnicodeStringPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringPrintfEx	The RtlUnicodeStringPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringValidate	The RtlUnicodeStringValidate function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringValidateEx	The RtlUnicodeStringValidateEx function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringVPrintf	The RtlUnicodeStringVPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringVPrintfEx	The RtlUnicodeStringVPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

Safe Integer Library Routines

This section describes the safe integer functions for drivers. These functions are defined as inline functions in the Ntintsafe.h header file in the WDK. The safe integer functions are designed to help drivers to avoid arithmetic overflow errors. These functions are divided into two sets; the first converts integer values from one type to another, and the second performs math functions. For more information about these functions, see Using Safe Integer Functions.

The Intsafe.h header file in the Windows SDK defines a similar set of safe integer functions for use by applications. For more information about this version of the safe integer functions, see Intsafe.h Functions.

Data Conversions

FUNCTION	DESCRIPTION
InterlockedExchange	Sets a variable of type LONG to a given value as an atomic operation; returns the original value of the variable.
RtlConvertLongToLargeInteger	Converts a given LONG value to a LARGE_INTEGER value.
RtlConvertUlongToLargeInteger	Converts a given ULONG value to a LARGE_INTEGER value.
RtlTimeFieldsToTime	Converts information in a TIME_FIELDS structure to system time.
RtlTimeToTimeFields	Converts a system time value into a buffered TIME_FIELDS value.
ExSystemTimeToLocalTime	Adds the time-zone bias for the current locale to GMT system time, converting it to local time.
ExLocalTimeToSystemTime	Subtracts the time-zone bias from the local time, converting it to GMT system time.
RtlAnsiStringToUnicodeString	Converts a buffered ANSI string to a Unicode string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)
RtlUnicodeStringToAnsiString	Converts a buffered Unicode string to an ANSI string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)
RtlUpperString	Converts a copy of a buffered string to uppercase and stores the copy in a destination buffer.
RtlUpcaseUnicodeString	Converts a copy of a buffered Unicode string to uppercase and stores the copy in a destination buffer.
RtlCharToInteger	Converts a single-byte character value into an integer in the specified base.
RtlIntegerToUnicodeString	Converts an unsigned integer value in the specified base to one or more Unicode characters in the given buffer.
RtlUnicodeStringToInteger	Converts a Unicode string representation of an integer into its integer equivalent.

Access to Driver-Managed Objects

FUNCTION	DESCRIPTION
ExCreateCallback	Creates or opens a callback object.
ExNotifyCallback	Calls the callback routines registered with a previously created or opened callback object.
ExRegisterCallback	Registers a callback routine with a previously created or opened callback object, so that the caller can be notified when conditions defined for the callback routine occur.
ExUnregisterCallback	Cancels the registration of a callback routine with a callback object.
IoRegisterDeviceInterface	Registers device functionality (a device interface) that a driver can enable for use by applications or other system components.
IoSetDeviceInterfaceState	Enables or disables a previously registered device interface. Applications and other system components can open only interfaces that are enabled.
IoGetDeviceInterfaceAlias	Returns the alias device interface of the specified interface class, if the alias exists. Device interfaces are considered aliases if they are exposed by the same underlying device and have identical interface reference strings, but are of different interface classes.
IoGetDeviceInterfaces	Returns a list of device interfaces of a particular device interface class (such as all devices on the system that support a HID interface).
IoGetFileObjectGenericMapping	Returns information about the mapping between generic access rights and specific access rights for file objects.
IoSetShareAccess	Sets the access allowed to a given file object representing a device. (Only highest-level drivers can call this routine.)
IoCheckShareAccess	Checks whether a request to open a file object specifies a desired access that is compatible with the current shared access permissions for the open file object. (Only highest-level drivers can call this routine.)
IoUpdateShareAccess	Modifies the current shared access permissions on the given file object. (Only highest-level drivers can call this routine.)
IoRemoveShareAccess	Restores the shared access permissions on the given file object that were modified by a preceding call to IoUpdateShareAccess.
RtlLengthSecurityDescriptor	Returns the size in bytes of a given security descriptor.
RtlValidSecurityDescriptor	Returns whether a given security descriptor is valid.
RtlCreateSecurityDescriptor	Initializes a new security descriptor to an absolute format with default values (in effect, with no security constraints).

FUNCTION	DESCRIPTION
RtlSetDaclSecurityDescriptor	Sets the discretionary ACL information for a given security descriptor in absolute format.
SeAssignSecurity	Builds a security descriptor for a new object, given the security descriptor of its parent directory (if any) and an originally requested security for the object.
SeDeassignSecurity	Deallocates the memory associated with a security descriptor that was created with SeAssignSecurity.
SeValidSecurityDescriptor	Returns whether a given security descriptor is structurally valid.
SeAccessCheck	Returns a Boolean value indicating whether the requested access rights can be granted to an object protected by a security descriptor and, possibly, a current owner.
SeSinglePrivilegeCheck	Returns a Boolean value indicating whether the current thread has at least the given privilege level.

Error Handling

FUNCTION	DESCRIPTION
IoAllocateErrorLogEntry	Allocates and initializes an error log packet; returns a pointer so the caller can supply error log data and call IoWriteErrorLogEntry with the packet.
IoFreeErrorLogEntry	Frees an error log entry allocated by IoAllocateErrorLogEntry.
IoWriteErrorLogEntry	Queues a previously allocated error log packet, filled in by the driver, to the system error logging thread.
IoIsErrorUserInduced	Returns a Boolean value indicating whether an I/O request failed due to one of the following (user-correctable) conditions: STATUS_IO_TIMEOUT, STATUS_DEVICE_NOT_READY, STATUS_UNRECOGNIZED_MEDIA, STATUS_VERIFY_REQUIRED, STATUS_WRONG_VOLUME, STATUS_MEDIA_WRITE_PROTECTED, or STATUS_NO_MEDIA_IN_DEVICE. If the result is TRUE, a removable-media driver must call IoSetHardErrorOrVerifyDevice before completing the IRP.
IoSetHardErrorOrVerifyDevice	Supplies the device object for which the given IRP was failed due to a user-induced error, such as supplying the incorrect media for the requested operation or changing the media before the requested operation was completed. (A file system driver uses the associated device object to send a dialog box to the user; the user can then correct the error or retry the operation.)
IoSetThreadHardErrorMode	Enables or disables error reporting for the current thread using IoRaiseHardError or IoRaiseInformationalHardError.

FUNCTION	DESCRIPTION
IoRaiseHardError	Causes a dialog box to be sent to the user indicating that the given IRP was failed on the given device object for an optional VPB, so that the user can correct the error or retry the operation.
IoRaiseInformationalHardError	Causes a dialog box to be sent to the user, showing an I/O error status and optional string supplying more information.
ExRaiseStatus	Raises an error status so that a caller-supplied structured exception handler is called. (This routine is useful only to highest-level drivers that supply exception handlers, in particular to file systems.)
KeBugCheckEx	Brings down the system in a controlled manner, displaying the bug-check code and possibly more information, after the caller discovers an unrecoverable inconsistency that will corrupt the system unless it is brought down. After the system is brought down, this routine displays bug-check and possibly other information. (This routine can be called when debugging under-development drivers. Otherwise, drivers should never call this routine when they can handle an error by failing an IRP and by calling IoAllocateErrorLogEntry and IoWriteErrorLogEntry.)
KeBugCheck	Brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that will corrupt the system if the caller continues to run. KeBugCheckEx is preferable.
KeInitializeCallbackRecord	Initializes a bug-check callback record before a device driver calls KeRegisterBugCheckCallback.
KeRegisterBugCheckCallback	Registers the device driver's bug-check callback routine, which is called if a system bug check occurs. Such a driver-supplied routine saves driver-determined state information, such as the contents of device registers, that would not otherwise be written into the system crash-dump file.
KeDeregisterBugCheckCallback	Removes a device driver's callback routine from the set of registered bug-check callback routines.

IOCTLs

TITLE	DESCRIPTION
IOCTL_SYSENV_ENUM_VARIABLES	Returns information about system environment variables using SysEnv device.
IOCTL_SYSENV_GET_VARIABLE	Gets the value of the specified system environment variables using SysEnv device.
IOCTL_SYSENV_QUERY_VARIABLE_INFO	Returns information about system environment variables using SysEnv device.

TITLE	DESCRIPTION
IOCTL_SYSENV_SET_VARIABLE	Sets the value of the specified system environment variables using SysEnv device.

Enumerations

TITLE	DESCRIPTION
BDCB_CALLBACK_TYPE	The BDCB_CALLBACK_TYPE enumeration specifies whether the callback being passed to a BOOT_DRIVER_CALLBACK_FUNCTION routine is a status update or a boot-start driver initialization notification.
BDCB_CLASSIFICATION	The BDCB_CLASSIFICATION enumeration lists different classifications of boot start images.
BDCB_STATUS_UPDATE_TYPE	The BDCB_STATUS_UPDATE_TYPE enumeration lists the types of boot-driver callback status updates.
BOUND_CALLBACK_STATUS	The BOUND_CALLBACK_STATUS enumeration indicates how a user-mode bounds exception was processed by the BoundCallback function.
BUS_DATA_TYPE	The BUS_DATA_TYPE enumeration indicates the type of bus configuration space.
BUS_DATA_TYPE	The BUS_DATA_TYPE enumeration indicates the type of bus configuration space.
CLFS_CONTEXT_MODE	The CLFS_CONTEXT_MODE enumeration indicates the type of sequence that the Common Log File System (CLFS) driver follows when it reads a set of records from a stream.
CLFS_MGMT_POLICY_TYPE	The CLFS_MGMT_POLICY_TYPE enumeration type identifies the type of a CLFS management policy.
CLS_LOG_INFORMATION_CLASS	The CLFS_LOG_INFORMATION_CLASS enumeration indicates the type of information that is requested by a call to ClfsQueryLogFileInformation.
D3COLD_LAST_TRANSITION_STATUS	The D3COLD_LAST_TRANSITION_STATUS enumeration indicates whether the most recent transition to the D3hot device power state was followed by a transition to the D3cold device power state.
DEVICE_INSTALL_STATE	The DEVICE_INSTALL_STATE enumeration describes a device's installation state.
DEVICE_POWER_STATE	The DEVICE_POWER_STATE enumeration type indicates a device power state.
DEVICE_POWER_STATE	The DEVICE_POWER_STATE enumeration type indicates a device power state.

TITLE	DESCRIPTION
DEVICE_REGISTRY_PROPERTY	The DEVICE_REGISTRY_PROPERTY enumeration identifies device properties that are stored in the registry.
DEVICE_REGISTRY_PROPERTY	The DEVICE_REGISTRY_PROPERTY enumeration identifies device properties that are stored in the registry.
DEVICE_REMOVAL_POLICY	The DEVICE_REMOVAL_POLICY enumeration describes a device's removal policy.
DEVICE_RESET_TYPE	The DEVICE_RESET_TYPE enumeration specifies the type of device reset that is being requested by a call to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
DEVICE_WAKE_DEPTH	The DEVICE_WAKE_DEPTH enumeration specifies the deepest device power state from which a device can trigger a wake signal.
DMA_COMPLETION_STATUS	The DMA_COMPLETION_STATUS enumeration describes the completion status of a DMA transfer.
ENLISTMENT_INFORMATION_CLASS	The ENLISTMENT_INFORMATION_CLASS enumeration identifies the type of information that the ZwSetInformationEnlistment routine can set and that the ZwQueryInformationEnlistment routine can retrieve for an enlistment object.
GPIO_PIN_CONFIG_TYPE	The GPIO_PIN_CONFIG_TYPE enumeration describes a connection IO resource.
GPIO_PIN_CONFIG_TYPE	The GPIO_PIN_CONFIG_TYPE enumeration describes a connection IO resource.
GPIO_PIN_IORESTRICTION_TYPE	The GPIO_PIN_IORESTRICTION_TYPE enumeration describes the functions that a GPIO pin is limited to performing.
GPIO_PIN_IORESTRICTION_TYPE	The GPIO_PIN_IORESTRICTION_TYPE enumeration describes the functions that a GPIO pin is limited to performing.
HARDWARE_COUNTER_TYPE	The HARDWARE_COUNTER_TYPE enumeration specifies the type of a hardware counter.
IMAGE_POLICY_ENTRY_TYPE	This enumeration is not supported.
IMAGE_POLICY_ID	This enumeration is not supported.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
IO_ACCESS_MODE	Defines the types of access mode for Scheduled File I/O (SFIO).

TITLE	DESCRIPTION
IO_ACCESS_MODE	Defines the types of access mode for Scheduled File I/O (SFIO).
IO_ACCESS_TYPE	Defines the access rights for Scheduled File I/O (SFIO).
IO_ACCESS_TYPE	Defines the access rights for Scheduled File I/O (SFIO).
IO_ALLOCATION_ACTION	The IO_ALLOCATION_ACTION enumerated type is used to specify return values for AdapterControl and ControllerControl routines.
IO_CONTAINER_INFORMATION_CLASS	The IO_CONTAINER_INFORMATION_CLASS enumeration contains constants that indicate the classes of system information that a kernel-mode driver can request.
IO_CONTAINER_NOTIFICATION_CLASS	The IO_CONTAINER_NOTIFICATION_CLASS enumeration contains constants that indicate the classes of events for which a kernel-mode driver can register to receive notifications.
IO_PAGING_PRIORITY	The IO_PAGING_PRIORITY enumeration describes the priority value for a paging I/O IRP.
IO_PRIORITY_HINT	The IO_PRIORITY_HINT enumeration type specifies the priority hint for an IRP.
IO_SESSION_EVENT	The IO_SESSION_EVENT enumeration indicates the type of session event for which a driver is receiving notification.
IO_SESSION_STATE	The IO_SESSION_STATE enumeration contains constants that indicate the current state of a user session.
IRQ_DEVICE_POLICY	The IRQ_DEVICE_POLICY enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.
IRQ_DEVICE_POLICY	The IRQ_DEVICE_POLICY enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.
IRQ_PRIORITY	The IRQ_PRIORITY enumeration type indicates the priority the system should give to servicing a device's interrupts.
IRQ_PRIORITY	The IRQ_PRIORITY enumeration type indicates the priority the system should give to servicing a device's interrupts.
KBUGCHECK_CALLBACK_REASON	The KBUGCHECK_CALLBACK_REASON enumeration type specifies the situations in which a bug-check callback executes.
KBUGCHECK_DUMP_IO_TYPE	The KBUGCHECK_DUMP_IO_TYPE enumeration type identifies the type of a section of data within a crash dump file.
KEY_INFORMATION_CLASS	The KEY_INFORMATION_CLASS enumeration type represents the type of information to supply about a registry key.

TITLE	DESCRIPTION
KEY_SET_INFORMATION_CLASS	The KEY_SET_INFORMATION_CLASS enumeration type represents the type of information to set for a registry key.
KEY_VALUE_INFORMATION_CLASS	The KEY_VALUE_INFORMATION_CLASS enumeration type specifies the type of information to supply about the value of a registry key.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
KINTERRUPT_POLARITY	The KINTERRUPT_POLARITY enumeration indicates how a device signals an interrupt request on an interrupt line.
KINTERRUPT_POLARITY	The KINTERRUPT_POLARITY enumeration indicates how a device signals an interrupt request on an interrupt line.
KTMOBJECT_TYPE	The KTMOBJECT_TYPE enumeration identifies the types of objects that KTM supports.
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.
MEMORY_INFORMATION_CLASS	Defines classes of memory information that can be retrieved by using the ZwQueryVirtualMemory function.
MONITOR_DISPLAY_STATE	Indicates the power state of the monitor being displayed on.
MONITOR_DISPLAY_STATE	Indicates the power state of the monitor being displayed on.
PEP_ACPI_OBJECT_TYPE	The PEP_ACPI_OBJECT_TYPE enumeration indicates the type of ACPI object.
PEP_ACPI_OBJECT_TYPE	The PEP_ACPI_OBJECT_TYPE enumeration indicates the type of ACPI object.
PEP_ACPI_RESOURCE_TYPE	The PEP_ACPI_RESOURCE_TYPE enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.

TITLE	DESCRIPTION
PEP_ACPI_RESOURCE_TYPE	The PEP_ACPI_RESOURCE_TYPE enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.
PEP_DEVICE_ACCEPTANCE_TYPE	The PEP_DEVICE_ACCEPTANCE_TYPE enumeration indicates whether a PEP accepts ownership of a device.
PEP_DEVICE_ACCEPTANCE_TYPE	The PEP_DEVICE_ACCEPTANCE_TYPE enumeration indicates whether a PEP accepts ownership of a device.
PEP_PERF_STATE_TYPE	The PEP_PERF_STATE_TYPE enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.
PEP_PERF_STATE_TYPE	The PEP_PERF_STATE_TYPE enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.
PEP_PERF_STATE_UNIT	The PEP_PERF_STATE_UNIT enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.
PEP_PERF_STATE_UNIT	The PEP_PERF_STATE_UNIT enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.
PEP_PROCESSOR_IDLE_CANCEL_CODE	The PEP_PROCESSOR_IDLE_CANCEL_CODE enumeration values indicate reasons why a processor cannot enter an idle state that was previously selected by the platform extension plug-in (PEP).
PEP_PROCESSOR_IDLE_TYPE	The PEP_PROCESSOR_IDLE_TYPE enumeration indicates whether idle constraints apply to just the current processor or to all processors in the hardware platform.
PEP_WORK_TYPE	The PEP_WORK_TYPE enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.
PEP_WORK_TYPE	The PEP_WORK_TYPE enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.
PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE	The PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE enumeration contains constants that indicate the type of hardware performance counter resource that is described by a PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure.
PO_FX_PERF_STATE_TYPE	The PO_FX_PERF_STATE_TYPE enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.
PO_FX_PERF_STATE_TYPE	The PO_FX_PERF_STATE_TYPE enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.

TITLE	DESCRIPTION
PO_FX_PERF_STATE_UNIT	The PO_FX_PERF_STATE_UNIT enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.
PO_FX_PERF_STATE_UNIT	The PO_FX_PERF_STATE_UNIT enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.
POOL_TYPE	The POOL_TYPE enumeration type specifies the type of system memory to allocate.
POOL_TYPE	The POOL_TYPE enumeration type specifies the type of system memory to allocate.
POWER_ACTION	The POWER_ACTION enumeration identifies the system power actions that can occur on a computer.
POWER_INFORMATION_LEVEL	Indicates power level information.
POWER_INFORMATION_LEVEL	Indicates power level information.
POWER_MONITOR_REQUEST_REASON	Defines values for power transitions reasons for a monitor.
POWER_REQUEST_TYPE	The POWER_REQUEST_TYPE enumeration indicates the power request type.
POWER_REQUEST_TYPE	The POWER_REQUEST_TYPE enumeration indicates the power request type.
POWER_STATE_TYPE	The POWER_STATE_TYPE enumeration type indicates that a power state value is a system power state or a device power state.
POWER_STATE_TYPE	The POWER_STATE_TYPE enumeration type indicates that a power state value is a system power state or a device power state.
PSCREATEPROCESSNOTIFYTYPE	Indicates the type of process notification. This enumeration is used in PsSetCreateProcessNotifyRoutineEx2 to register callback notifications.
PSCREATETHREADNOTIFYTYPE	Indicates the type of thread notification. This enumeration is used in PsSetCreateThreadNotifyRoutineEx to register callback notifications associated with thread creation or deletion.
REG_NOTIFY_CLASS	The REG_NOTIFY_CLASS enumeration type specifies the type of registry operation that the configuration manager is passing to a RegistryCallback routine.
RESOURCEMANAGER_INFORMATION_CLASS	The RESOURCEMANAGER_INFORMATION_CLASS enumeration identifies the type of information that the ZwQueryInformationResourceManager routine can retrieve for a resource manager object.

TITLE	DESCRIPTION
SUBSYSTEM_INFORMATION_TYPE	Indicates the type of subsystem for a process or thread. This enumeration is used in NtQueryInformationProcess and NtQueryInformationThread calls.
SYSTEM_POWER_STATE	The SYSTEM_POWER_STATE enumeration type is used to indicate a system power state.
SYSTEM_POWER_STATE	The SYSTEM_POWER_STATE enumeration type is used to indicate a system power state.
TRACE_INFORMATION_CLASS	The TRACE_INFORMATION_CLASS enumeration type is used to indicate types of information associated with a WMI event tracing session.
TRANSACTION_INFORMATION_CLASS	The TRANSACTION_INFORMATION_CLASS enumeration specifies the type of information that ZwSetInformationTransaction can set and ZwQueryInformationTransaction can retrieve for a transaction manager object.
TRANSACTION_OUTCOME	The TRANSACTION_OUTCOME enumeration defines the outcomes (results) that KTM can assign to a transaction.
TRANSACTION_STATE	The TRANSACTION_STATE enumeration defines the states that KTM can assign to a transaction.
TRANSACTIONMANAGER_INFORMATION_CLASS	The TRANSACTIONMANAGER_INFORMATION_CLASS enumeration specifies the type of information that the ZwQueryInformationTransactionManager routine can retrieve for a transaction manager object.
WORK_QUEUE_TYPE	The WORK_QUEUE_TYPE enumeration type indicates the type of system worker thread that handles a work item.

Functions

TITLE	DESCRIPTION
ALLOCATE_FUNCTION_EX	The LookasideListAllocateEx routine allocates the storage for a new lookaside-list entry when a client requests an entry from a lookaside list that is empty.
AppendTailList	The AppendTailList routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of LIST_ENTRY structures.
AppendTailList	The AppendTailList routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of LIST_ENTRY structures.
ARGUMENT_PRESENT	The ARGUMENT_PRESENT macro takes an argument pointer and returns FALSE if the pointer is NULL. Otherwise, it returns TRUE.

TITLE	DESCRIPTION
AuxKlibEnumerateSystemFirmwareTables	The AuxKlibEnumerateSystemFirmwareTables routine enumerates all system firmware tables of the specified type.
AuxKlibGetBugCheckData	The AuxKlibGetBugCheckData routine retrieves information about a bug check that has just occurred.
AuxKlibGetImageExportDirectory	The AuxKlibGetImageExportDirectory routine returns an image module's export directory.
AuxKlibGetSystemFirmwareTable	The AuxKlibGetSystemFirmwareTable routine retrieves the specified firmware table from the firmware table provider.
AuxKlibInitialize	The AuxKlibInitialize routine initializes the Auxiliary Kernel-Mode Library.
AuxKlibQueryModuleInformation	The AuxKlibQueryModuleInformation routine retrieves information about the image modules that the operating system has loaded.
BOUND_CALLBACK	The BoundCallback routine is executed whenever the system issues a bounds exception for a user-mode thread.
ClfsAddLogContainer	The ClfsAddLogContainer routine adds a container to a CLFS log.
ClfsAddLogContainerSet	The ClfsAddLogContainerSet routine atomically adds a set of containers to a CLFS log.
ClfsAdvanceLogBase	The ClfsAdvanceLogBase routine sets the base LSN of a CLFS stream.
ClfsAlignReservedLog	The ClfsAlignReservedLog routine calculates the size of the space that must be reserved for a specified set of records. The size calculation includes the space required for headers and the space required for sector alignment.
ClfsAllocReservedLog	The ClfsAllocReservedLog routine reserves space in a marshalling area for a set of records.
ClfsCloseAndResetLogFile	The ClfsCloseAndResetLogFile routine releases all references to a specified log file object and marks its associated stream for reset.
ClfsCloseLogFileObject	The ClfsCloseLogFileObject routine releases all references to a log file object.
ClfsCreateLogFile	The ClfsCreateLogFile routine creates or opens a CLFS stream. If necessary, ClfsCreateLogFile also creates the underlying physical log that holds the stream's records.
ClfsCreateMarshallingArea	The ClfsCreateMarshallingArea routine creates a marshalling area for a CLFS stream and returns a pointer to an opaque context that represents the new marshalling area.

TITLE	DESCRIPTION
ClfsCreateScanContext	The ClfsCreateScanContext routine creates a scan context that can be used to iterate over the containers of a specified CLFS log.
ClfsDeleteLogByPointer	The ClfsDeleteLogByPointer routine marks a CLFS stream for deletion.
ClfsDeleteLogFile	The ClfsDeleteLogFile routine marks a CLFS stream for deletion.
ClfsDeleteMarshallingArea	The ClfsDeleteMarshallingArea routine deletes a marshalling area.
ClfsFlushBuffers	The ClfsFlushBuffers routine forces all log I/O blocks in a specified marshalling area to stable storage.
ClfsFlushToLsn	The ClfsFlushToLsn routine forces, to stable storage, all records that have an LSN less than or equal to a specified LSN.
ClfsGetContainerName	The ClfsGetContainerName routine returns the path name of a specified container.
ClfsGetIoStatistics	The ClfsGetIoStatistics routine returns I/O statistics for a specified CLFS log.
ClfsLsnBlockOffset	The ClfsLsnBlockOffset routine returns the sector-aligned block offset contained in a specified LSN.
ClfsLsnContainer	The ClfsLsnContainer routine returns the logical container identifier contained in a specified LSN.
ClfsLsnCreate	The ClfsLsnCreate routine creates a log sequence number (LSN), given a container identifier, a block offset, and a record sequence number.
ClfsLsnEqual	The ClfsLsnEqual routine determines whether two LSNs from the same stream are equal.
ClfsLsnGreater	The ClfsLsnGreater routine determines whether one LSN is greater than another LSN. The two LSNs must be from the same stream.
ClfsLsnLess	The ClfsLsnLess routine determines whether one LSN is less than another LSN. The two LSNs must be from the same stream.
ClfsLsnNull	The ClfsLsnNull routine determines whether a specified LSN is equal to the smallest possible LSN, CLFS_LSN_NULL.
ClfsLsnRecordSequence	The ClfsLsnRecordSequence routine returns the record sequence number contained in a specified LSN.

TITLE	DESCRIPTION
ClfsMgmtDeregisterManagedClient	The ClfsMgmtDeregisterManagedClient routine removes the connection between a client and a log, so that the client no longer manages the log.
ClfsMgmtHandleLogFileFull	The ClfsMgmtHandleLogFileFull routine attempts to make more space available in a log. It might make more space available by adding containers to the log, or it might ask clients to move their log tails.
ClfsMgmtInstallPolicy	The ClfsMgmtInstallPolicy routine adds a CLFS_MGMT_POLICY structure to a physical log.
ClfsMgmtQueryPolicy	The ClfsMgmtQueryPolicy routine retrieves a specific CLFS_MGMT_POLICY structure for a log.
ClfsMgmtRegisterManagedClient	The ClfsMgmtRegisterManagedClient routine creates a client that will manage a CLFS log.
ClfsMgmtRemovePolicy	The ClfsMgmtRemovePolicy routine resets a log's CLFS_MGMT_POLICY structure to its default value.
ClfsMgmtSetLogFileSize	The ClfsMgmtSetLogFileSize routine adds containers to a log or deletes containers from a log.
ClfsMgmtSetLogFileSizeAsClient	The ClfsMgmtSetLogFileSizeAsClient routine sets the log file size by adding containers to a client log or deleting containers from a client log.
ClfsMgmtTailAdvanceFailure	The ClfsMgmtTailAdvanceFailure routine notifies CLFS management that the client could not advance the log's tail.
ClfsQueryLogFileInformation	The ClfsQueryLogFileInformation routine returns metadata and state information for a specified CLFS stream or its underlying physical log or both.
ClfsReadLogRecord	The ClfsReadLogRecord routine reads a target record in a CLFS stream and returns a read context that the caller can use to read the records preceding or following it in the stream.
ClfsReadNextLogRecord	The ClfsReadNextLogRecord routine reads the next record in a sequence, relative to the current record in a read context.
ClfsReadPreviousRestartArea	The ClfsReadPreviousRestartArea routine reads the previous restart record relative to the current record in a read context.
ClfsReadRestartArea	The ClfsReadRestartArea routine reads the restart record that was most recently written to a specified CLFS stream.
ClfsRemoveLogContainer	The ClfsRemoveLogContainer routine removes a container from a CLFS log.
ClfsRemoveLogContainerSet	The ClfsRemoveLogContainerSet routine atomically removes a set of containers from a CLFS log.

TITLE	DESCRIPTION
ClfsReserveAndAppendLog	The ClfsReserveAndAppendLog routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically.
ClfsReserveAndAppendLogAligned	The ClfsReserveAndAppendLogAligned routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically. The record's data is aligned on specified boundaries.
ClfsScanLogContainers	The ClfsScanLogContainers routine retrieves descriptive information for a sequence of containers that belong to a particular CLFS log.
ClfsSetArchiveTail	The ClfsSetArchiveTail routine sets the archive tail of a CLFS log to a specified LSN.
ClfsSetEndOfLog	The ClfsSetEndOfLog routine truncates a CLFS stream.
ClfsSetLogFileInformation	The ClfsSetLogFileInformation routine sets metadata and state information for a specified stream and its underlying physical log.
ClfsTerminateReadLog	The ClfsTerminateReadLog routine invalidates a specified read context after freeing resources associated with the context.
ClfsWriteRestartArea	The ClfsWriteRestartArea routine atomically appends a new restart record to a CLFS stream, flushes the restart record to stable storage, and optionally updates the base LSN of the stream.
CmCallbackGetKeyObjectID	The CmCallbackGetKeyObjectID routine retrieves the unique identifier and object name that are associated with a specified registry key object.
CmCallbackGetKeyObjectIDEx	The CmCallbackGetKeyObjectIDEx routine retrieves the unique identifier and object name that are associated with a specified registry key object.
CmCallbackReleaseKeyObjectIDEx	The CmCallbackReleaseKeyObjectIDEx routine frees an object name string obtained from the CmCallbackGetKeyObjectIDEx routine.
CmGetBoundTransaction	The CmGetBoundTransaction routine returns a pointer to the transaction object that represents the transaction, if any, that is associated with a specified registry key object.
CmGetCallbackVersion	The CmGetCallbackVersion routine retrieves the major and minor version numbers for the current version of the configuration manager's registry callback feature.
CmRegisterCallback	The CmRegisterCallback routine is obsolete for Windows Vista and later operating system versions. Use CmRegisterCallbackEx instead.The CmRegisterCallback routine registers a RegistryCallback routine.

TITLE	DESCRIPTION
CmRegisterCallbackEx	The CmRegisterCallbackEx routine registers a RegistryCallback routine.
CmSetCallbackObjectContext	The CmSetCallbackObjectContext routine associates specified context information with a specified registry object.
CmUnRegisterCallback	The CmUnRegisterCallback routine unregisters a RegistryCallback routine that a CmRegisterCallback or CmRegisterCallbackEx routine previously registered.
DMA_COMPLETION_ROUTINE	The DmaCompletionRoutine callback routine notifies the driver that previously requested a system DMA transfer that this transfer is complete.
DRIVER_ADD_DEVICE	The AddDevice routine is responsible for creating functional device objects (FDO) or filter device objects (filter DO) for devices enumerated by the Plug and Play (PnP) manager.
DRIVER_CANCEL	The Cancel routine cancels an I/O operation.
DRIVER_CONTROL	This routine starts a DMA data-transfer or a data transfer operation.
DRIVER_DISPATCH	The callback routine services various IRPs. For a list of function codes, see Remarks.
DRIVER_INITIALIZE	DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.
DRIVER_INITIALIZE	DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.
DRIVER_LIST_CONTROL	The AdapterListControl routine starts a direct memory access (DMA) scatter/gather operation.
DRIVER_REINITIALIZE	The Reinitialize routine continues driver and device initialization after the driver's DriverEntry routine returns.
DRIVER_STARTIO	The StartIo routine starts the I/O operation described by an IRP.
DRIVER_UNLOAD	The Unload routine performs any operations that are necessary before the system unloads the driver.
EX_CALLBACK_FUNCTION	A filter driver's RegistryCallback routine can monitor, block, or modify a registry operation.
ExAcquireResourceExclusiveLite	The ExAcquireResourceExclusiveLite routine acquires the given resource for exclusive access by the calling thread.
ExAcquireResourceSharedLite	The ExAcquireResourceSharedLite routine acquires the given resource for shared access by the calling thread.

TITLE	DESCRIPTION
ExAcquireRundownProtection	The ExAcquireRundownProtection routine tries to acquire run-down protection on a shared object so the caller can safely access the object.
ExAcquireRundownProtectionEx	The ExAcquireRundownProtectionEx routine tries to acquire run-down protection on a shared object so the caller can safely access the object.
ExAcquireSharedStarveExclusive	The ExAcquireSharedStarveExclusive routine acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.
ExAcquireSharedWaitForExclusive	The ExAcquireSharedWaitForExclusive routine acquires the given resource for shared access if shared access can be granted and there are no exclusive waiters.
ExAllocateFromLookasideListEx	The ExAllocateFromLookasideListEx routine removes the first entry from the specified lookaside list, or, if the list is empty, dynamically allocates the storage for a new entry.
ExAllocateFromNPagedLookasideList	The ExAllocateFromNPagedLookasideList routine returns a pointer to a nonpaged entry from the given lookaside list, or it returns a pointer to a newly allocated nonpaged entry.
ExAllocateFromPagedLookasideList	The ExAllocateFromPagedLookasideList routine returns a pointer to a paged entry from the given lookaside list, or it returns a pointer to a newly allocated paged entry.
ExAllocatePool	The ExAllocatePool routine is obsolete, and is exported only for existing binaries. Use ExAllocatePoolWithTag instead.ExAllocatePool allocates pool memory of the specified type and returns a pointer to the allocated block.
ExAllocatePoolWithQuota	The ExAllocatePoolWithQuota routine is obsolete, and is exported only for existing driver binaries. Use ExAllocatePoolWithQuotaTag instead.ExAllocatePoolWithQuota allocates pool memory, charging quota against the current process.
ExAllocatePoolWithQuotaTag	The ExAllocatePoolWithQuotaTag routine allocates pool memory, charging the quota against the current process.
ExAllocatePoolWithTag	The ExAllocatePoolWithTag routine allocates pool memory of the specified type and returns a pointer to the allocated block.
ExAllocatePoolWithTagPriority	The ExAllocatePoolWithTagPriority routine allocates pool memory of the specified type.
ExAllocateTimer	The ExAllocateTimer routine allocates and initializes a timer object.
ExCancelTimer	The ExCancelTimer routine cancels a timer that was set by a previous call to the ExSetTimer routine.

TITLE	DESCRIPTION
ExConvertExclusiveToSharedLite	The ExConvertExclusiveToSharedLite routine converts a given resource from acquired for exclusive access to acquired for shared access.
ExCreateCallback	The ExCreateCallback routine either creates a new callback object or opens an existing callback object on behalf of the caller.
ExDeleteLookasideListEx	The ExDeleteLookasideListEx routine deletes a lookaside list.
ExDeleteNPagedLookasideList	The ExDeleteNPagedLookasideList routine destroys a nonpaged lookaside list.
ExDeletePagedLookasideList	The ExDeletePagedLookasideList routine destroys a paged lookaside list.
ExDeleteResourceLite	The ExDeleteResourceLite routine deletes a given resource from the system's resource list.
ExDeleteTimer	The ExDeleteTimer routine deletes a timer object that was previously allocated by the ExAllocateTimer routine.
ExFlushLookasideListEx	The ExFlushLookasideListEx routine flushes all entries from the specified lookaside list and frees the allocated storage for each entry.
ExFreePool	The ExFreePool routine deallocates a block of pool memory.
ExFreePool	The ExFreePool routine deallocates a block of pool memory.
ExFreePoolWithTag	The ExFreePoolWithTag routine deallocates a block of pool memory allocated with the specified tag.
ExFreeToLookasideListEx	The ExFreeToLookasideListEx routine inserts an entry into a lookaside list, or, if the list is full, frees the allocated storage for the entry.
ExFreeToNPagedLookasideList	The ExFreeToNPagedLookasideList routine returns a nonpaged entry to the given lookaside list or to nonpaged pool.
ExFreeToPagedLookasideList	The ExFreeToPagedLookasideList routine returns a pageable entry to the given lookaside list or to paged pool.
ExGetExclusiveWaiterCount	The ExGetExclusiveWaiterCount routine returns the number of waiters on exclusive access to a given resource.
ExGetFirmwareEnvironmentVariable	The ExGetFirmwareEnvironmentVariable routine gets the value of the specified system firmware environment variable.
ExGetFirmwareType	Returns the system firmware type.
ExGetPreviousMode	The ExGetPreviousMode routine returns the previous processor mode for the current thread.

TITLE	DESCRIPTION
ExGetSharedWaiterCount	The ExGetSharedWaiterCount routine returns the number of waiters on shared access to a given resource.
ExInitializeDeleteTimerParameters	The ExInitializeDeleteTimerParameters routine initializes an EXT_DELETE_PARAMETERS structure.
ExInitializeFastMutex	The ExInitializeFastMutex routine initializes a fast mutex variable, used to synchronize mutually exclusive access by a set of threads to a shared resource.
ExInitializeLookasideListEx	The ExInitializeLookasideListEx routine initializes a lookaside list.
ExInitializeNPagedLookasideList	The ExInitializeNPagedLookasideList routine initializes a lookaside list for nonpaged entries of the specified size.
ExInitializePagedLookasideList	The ExInitializePagedLookasideList routine initializes a lookaside list for pageable entries of the specified size.
ExInitializeResourceLite	The ExInitializeResourceLite routine initializes a resource variable.
ExInitializeRundownProtection	The ExInitializeRundownProtection routine initializes run-down protection on a shared object.
ExInitializeSetTimerParameters	The ExInitializeSetTimerParameters routine initializes an EXT_SET_PARAMETERS structure.
ExInterlockedAddLargeInteger	The ExInterlockedAddLargeInteger routine adds a large integer value to the specified variable as an atomic operation.
ExInterlockedAddLargeStatistic	The ExInterlockedAddLargeStatistic routine performs an interlocked addition of a ULONG increment value to a LARGE_INTEGER variable.
ExInterlockedAddUlong	The ExInterlockedAddUlong routine adds an unsigned long value to a given unsigned integer as an atomic operation.
ExInterlockedCompareExchange64	The ExInterlockedCompareExchange64 routine compares one integer variable to another and, if they are equal, sets the first variable to a caller-supplied value.
ExInterlockedFlushSList	The ExInterlockedFlushSList routine atomically removes all entries from a sequenced singly linked list.
ExInterlockedInsertHeadList	The ExInterlockedInsertHeadList routine atomically inserts an entry at the beginning of a doubly linked list of LIST_ENTRY structures.
ExInterlockedInsertTailList	The ExInterlockedInsertTailList routine atomically inserts an entry at the end of a doubly linked list of LIST_ENTRY structures.

TITLE	DESCRIPTION
ExInterlockedPopEntryList	The ExInterlockedPopEntryList routine atomically removes an entry from the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
ExInterlockedPopEntrySList	The ExInterlockedPopEntrySList routine atomically removes the first entry from a sequenced singly linked list.
ExInterlockedPushEntryList	The ExInterlockedPushEntryList routine atomically inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
ExInterlockedPushEntrySList	The ExInterlockedPushEntrySList routine atomically inserts an entry at the beginning of a sequenced singly linked list.
ExInterlockedRemoveHeadList	The ExInterlockedRemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
ExIsProcessorFeaturePresent	The ExIsProcessorFeaturePresent routine queries for the existence of a specified processor feature.
ExIsResourceAcquiredExclusiveLite	The ExIsResourceAcquiredExclusiveLite routine returns whether the current thread has exclusive access to a given resource.
ExIsResourceAcquiredSharedLite	The ExIsResourceAcquiredSharedLite routine returns whether the current thread has access (either shared or exclusive) to a given resource.
ExIsSoftBoot	Determines whether the system has gone through a soft restart.
ExLocalTimeToSystemTime	The ExLocalTimeToSystemTime routine converts a system time value for the current time zone to an unbiased, GreenGMT value.
ExNotifyCallback	The ExNotifyCallback routine causes all callback routines registered for the given object to be called.
EXPAND_STACK_CALLOUT	The ExpandedStackCall routine executes with a guaranteed stack size.
ExQueryDepthSList	The ExQueryDepthSList routine returns the number of entries currently in a given sequenced singly linked list.
ExQueryTimerResolution	The ExQueryTimerResolution routine reports the range of timer resolutions that are supported by the system clock.
ExRaiseAccessViolation	The ExRaiseAccessViolation routine can be used with structured exception handling to throw a driver-determined exception for a memory access violation that occurs when a driver processes I/O requests.

TITLE	DESCRIPTION
ExRaiseDatatypeMisalignment	The ExRaiseDatatypeMisalignment routine can be used with structured exception handling to throw a driver-determined exception for a misaligned data type that occurs when a driver processes I/O requests.
ExRaiseStatus	The ExRaiseStatus routine is called by drivers that supply structured exception handlers to handle particular errors that occur while they are processing I/O requests.
ExRegisterCallback	The ExRegisterCallback routine registers a given callback routine with a given callback object.
ExReinitializeResourceLite	The ExReinitializeResourceLite routine reinitializes an existing resource variable.
ExReInitializeRundownProtection	The ExReInitializeRundownProtection routine reinitializes an EX_RUNDOWN_REF structure after the associated object is run down.
ExReleaseResourceForThreadLite	The ExReleaseResourceForThreadLite routine releases the input resource of the indicated thread.
ExReleaseResourceLite	The ExReleaseResourceLite routine releases a specified executive resource owned by the current thread.
ExReleaseRundownProtection	The ExReleaseRundownProtection routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtection routine.
ExReleaseRundownProtectionEx	The ExReleaseRundownProtectionEx routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtectionEx routine.
ExReleaseSpinLockExclusive	The ExReleaseSpinLockExclusive routine releases a spin lock that the caller previously acquired for exclusive access, and restores the IRQL to its original value.
ExReleaseSpinLockShared	The ExReleaseSpinLockShared routine releases ownership of a spin lock that the caller previously acquired for shared access, and restores the IRQL to its original value.
ExRundownCompleted	The ExRundownCompleted routine updates the run-down status of a shared object to indicate that the run down of the object has completed.
ExSetFirmwareEnvironmentVariable	The ExSetFirmwareEnvironmentVariable routine sets the value of the specified system firmware environment variable.
ExSetResourceOwnerPointer	The ExSetResourceOwnerPointer routine sets the owner thread pointer for an executive resource.
ExSetResourceOwnerPointerEx	The ExSetResourceOwnerPointerEx routine transfers the ownership of an executive resource from the calling thread to an owner pointer, which is a system address that identifies the resource owner.

TITLE	DESCRIPTION
ExSetTimer	The ExSetTimer routine starts a timer operation and sets the timer to expire at the specified due time.
ExSetTimerResolution	The ExSetTimerResolution routine modifies the frequency at which the system clock interrupts. Use this routine with extreme caution (see the following Remarks section).
ExSystemTimeToLocalTime	The ExSystemTimeToLocalTime routine converts a GMT system time value to the local system time for the current time zone.
EXT_CALLBACK	An ExTimerCallback callback routine runs after an EX_TIMER timer object's time interval expires.
EXT_DELETE_CALLBACK	An ExTimerDeleteCallback callback routine runs when the operating system deletes an EX_TIMER timer object.
ExTryConvertSharedSpinLockExclusive	The ExTryConvertSharedSpinLockExclusive routine attempts to convert the access state of a spin lock from acquired for shared access to exclusive access.
ExUnregisterCallback	The ExUnregisterCallback routine removes a callback routine previously registered with a callback object from the list of routines to be called during the notification process.
ExUuidCreate	The ExUuidCreate routine initializes a UUID (GUID) structure to a newly generated value.
ExWaitForRundownProtectionRelease	The ExWaitForRundownProtectionRelease routine waits until all drivers that have already been granted run-down protection complete their accesses of the shared object.
FIELD_OFFSET	The FIELD_OFFSET macro returns the byte offset of a named field in a known structure type.
FIELD_OFFSET	The FIELD_OFFSET macro returns the byte offset of a named field in a known structure type.
FirstEntrySList	The FirstEntrySList routine returns the first entry in a sequenced singly linked list.
FREE_FUNCTION_EX	The LookasideListFreeEx routine frees the storage for a lookaside-list entry when a client tries to insert the entry into a lookaside list that is full.
GET_D3COLD_CAPABILITY	The GetBusDriverD3ColdSupport routine enables the driver for a device to query whether the enumerating bus driver supports the D3cold device power state.
GET_D3COLD_LAST_TRANSITION_STATUS	The GetLastTransitionStatus routine enables the driver for a device to query whether the most recent transition to the D3hot substate was followed by a transition to the D3cold substate.
GET_DMA_ADAPTER	The GetDmaAdapter routine returns a DMA_ADAPTER structure for the target device.

TITLE	DESCRIPTION
GET_IDLE_WAKE_INFO	The GetIdleWakeInfo routine enables the driver for a device to discover the device power states from which the device can signal a wake event.
GET_SDEV_IDENTIFIER	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
GET_SET_DEVICE_DATA	The GetBusData routine reads data from the device's configuration space.
HalAllocateHardwareCounters	The HalAllocateHardwareCounters routine allocates a set of hardware performance counters.
HalExamineMBR	The HalExamineMBR routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.
HalFreeHardwareCounters	The HalFreeHardwareCounters routine frees a set of hardware performance counters that was acquired in a previous call to HalAllocateHardwareCounters routine.
InitializeListHead	The InitializeListHead routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.
InitializeListHead	The InitializeListHead routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.
InitializeObjectAttributes	The InitializeObjectAttributes macro initializes the opaque OBJECT_ATTRIBUTES structure, which specifies the properties of an object handle to routines that open handles.
InitializeSListHead	The InitializeSListHead routine (or ExInitializeSListHead) initializes an SLIST_HEADER structure that represents the head of a sequenced singly linked list.
InsertHeadList	The InsertHeadList routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.
InsertHeadList	The InsertHeadList routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.
InsertTailList	The InsertTailList routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.
InsertTailList	The InsertTailList routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.
InterlockedAnd	The InterlockedAnd macro atomically computes a bitwise AND operation.

TITLE	DESCRIPTION
InterlockedAnd	The InterlockedAnd macro atomically computes a bitwise AND operation.
InterlockedCompareExchange	The InterlockedCompareExchange routine performs an atomic operation that compares the input value pointed to by Destination with the value of Comparand.
InterlockedCompareExchange	The InterlockedCompareExchange routine performs an atomic operation that compares the input value pointed to by Destination with the value of Comperand.
InterlockedCompareExchangePointer	The InterlockedCompareExchangePointer routine performs an atomic operation that compares the input pointer value pointed to by Destination with the pointer value Comparand.
InterlockedCompareExchangePointer	The InterlockedCompareExchangePointer routine performs an atomic operation that compares the input pointer value pointed to by Destination with the pointer value Comperand.
InterlockedDecrement	The InterlockedDecrement routine decrements a caller-supplied variable of type LONG as an atomic operation.
InterlockedDecrement	The InterlockedDecrement routine decrements a caller-supplied variable of type LONG as an atomic operation.
InterlockedExchange	The InterlockedExchange routine sets an integer variable to a given value as an atomic operation.
InterlockedExchange	The InterlockedExchange routine sets an integer variable to a given value as an atomic operation.
InterlockedExchangeAdd	The InterlockedExchangeAdd routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.
InterlockedExchangeAdd	The InterlockedExchangeAdd routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.
InterlockedExchangePointer	The InterlockedExchangePointer routine performs an atomic operation that sets a pointer to a new value.
InterlockedExchangePointer	The InterlockedExchangePointer routine performs an atomic operation that sets a pointer to a new value.
InterlockedIncrement	The InterlockedIncrement routine increments a caller-supplied variable as an atomic operation.
InterlockedIncrement	The InterlockedIncrement routine increments a caller-supplied variable as an atomic operation.
InterlockedOr	The InterlockedOr routine atomically computes a bitwise OR operation.

TITLE	DESCRIPTION
InterlockedOr	The InterlockedOr routine atomically computes a bitwise OR operation.
InterlockedXor	The InterlockedOr routine atomically computes a bitwise exclusive OR operation.
InterlockedXor	The InterlockedOr routine atomically computes a bitwise exclusive OR operation.
IO_COMPLETION_ROUTINE	The IoCompletion routine completes the processing of I/O operations.
IO_CSQ_ACQUIRE_LOCK	The CsqAcquireLock routine is used by the system to acquire the lock for a driver-implemented, cancel-safe IRP queue.
IO_CSQ_COMPLETE_CANCELED_IRP	The CsqCompleteCanceledIrp routine is used by the system to signal to the driver that it can complete a canceled IRP.
IO_CSQ_INSERT_IRP	The CsqInsertIrp routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.
IO_CSQ_INSERT_IRP_EX	The CsqInsertIrpEx routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.
IO_CSQ_PEEK_NEXT_IRP	The CsqPeekNextIrp routine is used by the system to find the next matching IRP in a driver-implemented, cancel-safe IRP queue.
IO_CSQ_RELEASE_LOCK	The CsqReleaseLock routine is used by the system to release the lock that was acquired using CsqAcquireLock.
IO_CSQ_REMOVE_IRP	The CsqRemoveIrp routine is used by the system to remove the specified IRP from a driver-implemented, cancel-safe IRP queue.
IO_DPC_ROUTINE	The DpcForIsr routine finishes the servicing of an I/O operation, after an InterruptService routine returns.
IO_SESSION_NOTIFICATION_FUNCTION	The IO_SESSION_NOTIFICATION_FUNCTION function type defines a callback routine through which a driver receives notifications of changes in the state of user sessions that the driver is interested in.
IO_TIMER_ROUTINE	The IoTimer routine is a DPC that, if registered, is called once per second.
IO_WORKITEM_ROUTINE	A WorkItem routine performs the processing for a work item that was queued by the IoQueueWorkItem routine.
IO_WORKITEM_ROUTINE_EX	A WorkItemEx routine performs the processing for a work item that was queued by the IoQueueWorkItemEx or IoTryQueueWorkItem routine.

TITLE	DESCRIPTION
IoAcquireRemoveLock	The IoAcquireRemoveLock routine increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack or deleted.
IoAdjustPagingPathCount	The IoAdjustPagingPathCount routine increments or decrements a caller-supplied page-file counter as an atomic operation.
IoAllocateController	The IoAllocateController routine sets up the call to a driver-supplied ControllerControl routine as soon as the device controller, represented by the given controller object, is available to carry out an I/O operation for the target device, represented by the given device object.
IoAllocateDriverObjectExtension	The IoAllocateDriverObjectExtension routine allocates a per-driver context area, called a driver object extension, and assigns a unique identifier to it.
IoAllocateErrorLogEntry	The IoAllocateErrorLogEntry routine allocates an error log entry, and returns a pointer to the packet that the caller uses to supply information about an I/O error.
IoAllocateIrp	The IoAllocateIrp routine allocates an IRP, given the number of I/O stack locations for each driver layered under the caller, and, optionally, for the caller.
IoAllocateMdl	The IoAllocateMdl routine allocates a memory descriptor list (MDL) large enough to map a buffer, given the buffer's starting address and length. Optionally, this routine associates the MDL with an IRP.
IoAllocateWorkItem	The IoAllocateWorkItem routine allocates a work item.
IoAssignArcName	The IoAssignArcName routine creates a symbolic link between the ARC name of a physical device and the name of the corresponding device object when it has been created.
IoAttachDevice	The IoAttachDevice routine attaches the caller's device object to a named target device object, so that I/O requests bound for the target device are routed first to the caller.
IoAttachDeviceToDeviceStack	The IoAttachDeviceToDeviceStack routine attaches the caller's device object to the highest device object in the chain and returns a pointer to the previously highest device object.
IoBuildAsynchronousFsdRequest	The IoBuildAsynchronousFsdRequest routine allocates and sets up an IRP to be sent to lower-level drivers.
IoBuildDeviceIoControlRequest	The IoBuildDeviceIoControlRequest routine allocates and sets up an IRP for a synchronously processed device control request.
IoBuildPartialMdl	The IoBuildPartialMdl routine builds a new memory descriptor list (MDL) that represents part of a buffer that is described by an existing MDL.

TITLE	DESCRIPTION
IoBuildSynchronousFsdRequest	The IoBuildSynchronousFsdRequest routine allocates and sets up an IRP for a synchronously processed I/O request.
IoCallDriver	The IoCallDriver routine sends an IRP to the driver associated with a specified device object.
IoCancelIrp	The IoCancelIrp routine sets the cancel bit in a given IRP and calls the cancel routine for the IRP if there is one.
IoCheckLinkShareAccess	The IoCheckLinkShareAccess routine is called by file system drivers (FSDs) or other highest-level drivers to check whether link shared access to a file object is permitted.
IoCheckShareAccess	The IoCheckShareAccess routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.
IoCheckShareAccessEx	The IoCheckShareAccessEx routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.
IoClearActivityIdThread	The IoClearActivityIdThread routine clears the activity ID of the current thread.
IoCompleteRequest	The IoCompleteRequest routine indicates that the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O manager.
IoConnectInterrupt	The IoConnectInterrupt routine registers a device driver's InterruptService routine (ISR), so that it will be called when a device interrupts on any of a specified set of processors.
IoConnectInterruptEx	For more information, see the WdmlibIoConnectInterruptEx function.#define IoConnectInterruptEx WdmlibIoConnectInterruptEx
IoCopyCurrentIrpStackLocationToNext	The IoCopyCurrentIrpStackLocationToNext routine copies the IRP stack parameters from the current I/O stack location to the stack location of the next-lower driver.
IoCreateController	The IoCreateController routine allocates memory for and initializes a controller object with a controller extension of a driver-determined size.
IoCreateDevice	The IoCreateDevice routine creates a device object for use by a driver.
IoCreateFile	The IoCreateFile routine either causes a new file or directory to be created, or it opens an existing file, device, directory, or volume, giving the caller a handle for the file object.
IoCreateNotificationEvent	The IoCreateNotificationEvent routine creates or opens a named notification event used to notify one or more threads of execution that an event has occurred.

TITLE	DESCRIPTION
IoCreateSymbolicLink	The IoCreateSymbolicLink routine sets up a symbolic link between a device object name and a user-visible name for the device.
IoCreateSynchronizationEvent	The IoCreateSynchronizationEvent routine creates or opens a named synchronization event for use in serialization of access to hardware between two otherwise unrelated drivers.
IoCreateSystemThread	The IoCreateSystemThread routine creates a system thread that executes in kernel mode, and supplies a handle for the thread.
IoCreateUnprotectedSymbolicLink	The IoCreateUnprotectedSymbolicLink routine sets up an unprotected symbolic link between a device object name and a corresponding Win32-visible name.
IoCsqInitialize	The IoCsqInitialize routine initializes the driver's cancel-safe IRP queue dispatch table.
IoCsqInitializeEx	The IoCsqInitializeEx routine initializes the dispatch table for a cancel-safe IRP queue.
IoCsqInsertIrp	The IoCsqInsertIrp routine inserts an IRP in the driver's cancel-safe IRP queue.
IoCsqInsertIrpEx	The IoCsqInsertIrpEx routine inserts an IRP into the driver's cancel-safe IRP queue.
IoCsqRemoveIrp	The IoCsqRemoveIrp routine removes a particular IRP from the queue.
IoCsqRemoveNextIrp	The IoCsqRemoveNextIrp routine removes the next matching IRP in the queue.
IoDeassignArcName	The IoDeassignArcName routine removes a symbolic link between the ARC name for a device and the named device object.
IoDecrementKeepAliveCount	The IoDecrementKeepAliveCount routine decrements a reference count associated with an Windows app on a specific device.
IoDeleteController	The IoDeleteController routine removes a given controller object from the system, for example, when the driver that created it is being unloaded.
IoDeleteDevice	The IoDeleteDevice routine removes a device object from the system, for example, when the underlying device is removed from the system.
IoDeleteSymbolicLink	The IoDeleteSymbolicLink routine removes a symbolic link from the system.
IoDetachDevice	The IoDetachDevice routine releases an attachment between the caller's device object and a lower driver's device object.

TITLE	DESCRIPTION
IoDisconnectInterrupt	The IoDisconnectInterrupt routine releases a device driver's set of interrupt object(s) when the device is paused or removed, or when the driver is being unloaded.
IoDisconnectInterruptEx	For more information, see the WdmlibIoDisconnectInterruptEx function.#define IoDisconnectInterruptEx WdmlibIoDisconnectInterruptEx
IoForwardIrpSynchronously	The IoForwardIrpSynchronously routine sends an IRP to a specified driver and waits for that driver to complete the IRP.
IoFreeController	The IoFreeController routine releases a previously allocated controller object when the driver has completed an I/O request.
IoFreeErrorLogEntry	The IoFreeErrorLogEntry routine frees an unused error log entry.
IoFreeIrp	The IoFreeIrp routine releases a caller-allocated IRP from the caller's IoCompletion routine.
IoFreeMdl	The IoFreeMdl routine releases a caller-allocated memory descriptor list (MDL).
IoFreeWorkItem	The IoFreeWorkItem routine frees a work item that was allocated by IoAllocateWorkItem.
IoGetActivityIdIrp	The IoGetActivityIdIrp routine retrieves the current activity ID associated with an IRP.
IoGetActivityIdThread	The IoGetActivityIdThread routine returns the activity ID associated with the current thread.
IoGetAffinityInterrupt	For more information, see the WdmlibIoGetAffinityInterrupt function.#define IoGetAffinityInterrupt WdmlibIoGetAffinityInterrupt
IoGetAttachedDeviceReference	The IoGetAttachedDeviceReference routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.
IoGetAttachedDeviceReference	The IoGetAttachedDeviceReference routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.
IoGetBootDiskInformation	The IoGetBootDiskInformation routine returns information describing the boot and system disks.

TITLE	DESCRIPTION
IoGetConfigurationInformation	The IoGetConfigurationInformation routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.
IoGetConfigurationInformation	The IoGetConfigurationInformation routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.
IoGetContainerInformation	The IoGetContainerInformation routine provides information about the current state of a user session.
IoGetCurrentIrpStackLocation	The IoGetCurrentIrpStackLocation routine returns a pointer to the caller's I/O stack location in the specified IRP.
IoGetCurrentProcess	The IoGetCurrentProcess routine returns a pointer to the current process.
IoGetDeviceInterfaceAlias	The IoGetDeviceInterfaceAlias routine returns the alias device interface of the specified device interface instance, if the alias exists.
IoGetDeviceInterfacePropertyData	The IoGetDeviceInterfacePropertyData routine retrieves the current value of a device interface property.
IoGetDeviceInterfaces	The IoGetDeviceInterfaces routine returns a list of device interface instances of a particular device interface class (such as all devices on the system that support a HID interface).
IoGetDeviceNumaNode	The IoGetDeviceNumaNode routine gets the node number of a device.
IoGetDeviceObjectPointer	The IoGetDeviceObjectPointer routine returns a pointer to the top object in the named device object's stack and a pointer to the corresponding file object, if the requested access to the objects can be granted.
IoGetDeviceProperty	The IoGetDeviceProperty routine retrieves information about a device such as configuration information and the name of its PDO.
IoGetDevicePropertyData	The IoGetDevicePropertyData routine retrieves the current setting for a device property.
IoGetDmaAdapter	The IoGetDmaAdapter routine returns a pointer to the DMA adapter structure for a physical device object.
IoGetDriverObjectExtension	The IoGetDriverObjectExtension routine retrieves a previously allocated per-driver context area.

TITLE	DESCRIPTION
IoGetFileObjectGenericMapping	The IoGetFileObjectGenericMapping routine returns information about the mapping between each generic access right and the set of specific access rights for file objects.
IoGetFunctionCodeFromCtlCode	The IoGetFunctionCodeFromCtlCode macro returns the value of the function code contained in an I/O control code.
IoGetInitialStack	The IoGetInitialStack routine returns the base address of the current thread's stack.
IoGetInitiatorProcess	The IoGetInitiatorProcess routine retrieves the process which initiated the creation of a file object if different than the process which is issuing the create.
IoGetIoPriorityHint	The IoGetIoPriorityHint routine gets the priority hint value from an IRP.
IoGetNextIrpStackLocation	The IoGetNextIrpStackLocation routine gives a higher level driver access to the next-lower driver's I/O stack location in an IRP so the caller can set it up for the lower driver.
IoGetPagingIoPriority	The IoGetPagingIoPriority routine indicates the priority level of a paging I/O request.
IoGetRelatedDeviceObject	Given a file object, the IoGetRelatedDeviceObject routine returns a pointer to the corresponding device object.
IoGetRemainingStackSize	The IoGetRemainingStackSize routine returns the current amount of available kernel-mode stack space.
IoGetStackLimits	The IoGetStackLimits routine returns the boundaries of the current thread's stack frame.
IoIncrementKeepAliveCount	The IoIncrementKeepAliveCount routine increments a reference count associated with an Windows app process on a specific device.
IoInitializeDpcRequest	The IoInitializeDpcRequest routine registers a driver-supplied DpcForIsr routine.
IoInitializeIrp	The IoInitializeIrp routine initializes a given IRP that was allocated by the caller.
IoInitializeRemoveLock	The IoInitializeRemoveLock routine initializes a remove lock for a device object.
IoInitializeTimer	The IoInitializeTimer routine sets up a driver-supplied IoTimer routine associated with a given device object.
IoInitializeWorkItem	The IoInitializeWorkItem routine initializes a work item that the caller has already allocated.

TITLE	DESCRIPTION
IoInvalidateDeviceRelations	The IoInvalidateDeviceRelations routine notifies the PnP manager that the relations for a device (such as bus relations, ejection relations, removal relations, and the target device relation) have changed.
IoInvalidateDeviceState	The IoInvalidateDeviceState routine notifies the PnP manager that some aspect of the PnP state of a device has changed.
IoIs32bitProcess	The IoIs32bitProcess routine checks whether the originator of the current I/O request is a 32-bit user-mode application.
IoIsErrorUserInduced	The IoIsErrorUserInduced routine determines whether an I/O error encountered while processing a request to a removable-media device was caused by the user.
IoIsValidIrpStatus	The IoIsValidIrpStatus routine validates the specified NTSTATUS status code value.
IoIsWdmVersionAvailable	The IoIsWdmVersionAvailable routine checks whether a given WDM version is supported by the operating system.
IoMakeAssociatedIrp	This routine is reserved for use by file systems and file system filter drivers.
IoMarkIrpPending	The IoMarkIrpPending routine marks the specified IRP, indicating that a driver's dispatch routine subsequently returned STATUS_PENDING because further processing is required by other driver routines.
IoOpenDeviceInterfaceRegistryKey	The IoOpenDeviceInterfaceRegistryKey routine returns a handle to a registry key for storing information about a particular device interface instance.
IoOpenDeviceRegistryKey	The IoOpenDeviceRegistryKey routine returns a handle to a device-specific or a driver-specific registry key for a particular device instance.
IoPropagateActivityIdToThread	The IoPropagateActivityIdToThread routine associates the activity ID from an IRP with the current thread.
IoQueryFullDriverPath	The IoQueryFullDriverPath routine retrieves the full path name of the binary file that is loaded for the specified driver object.
IoQueueWorkItem	The IoQueueWorkItem routine associates a WorkItem routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.
IoQueueWorkItemEx	The IoQueueWorkItemEx routine associates a WorkItemEx routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.
IoRaiseHardError	The IoRaiseHardError routine causes a dialog box to appears that warns the user that a device I/O error has occurred, which might indicate that a physical device is failing.

TITLE	DESCRIPTION
IoRaiseInformationalHardError	The IoRaiseInformationalHardError routine sends a dialog box to the user, warning about a device I/O error that indicates why a user I/O request failed.
IoRegisterBootDriverCallback	The IoRegisterBootDriverCallback routine registers a BOOT_DRIVER_CALLBACK_FUNCTION routine to be called during the initialization of a boot-start driver and its dependent DLLs.
IoRegisterBootDriverReinitialization	The IoRegisterBootDriverReinitialization routine is called by a boot driver to register the driver's reinitialization routine with the I/O manager to be called after all devices have been enumerated and started.
IoRegisterContainerNotification	The IoRegisterContainerNotification routine registers a kernel-mode driver to receive notifications about a specified class of events.
IoRegisterDeviceInterface	The IoRegisterDeviceInterface routine registers a device interface class, if it has not been previously registered, and creates a new instance of the interface class, which a driver can subsequently enable for use by applications or other system components.
IoRegisterDriverReinitialization	The IoRegisterDriverReinitialization routine is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.
IoRegisterLastChanceShutdownNotification	The IoRegisterLastChanceShutdownNotification routine registers a driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down, after all file systems have been flushed.
IoRegisterPlugPlayNotification	The IoRegisterPlugPlayNotification routine registers a Plug and Play (PnP) notification callback routine to be called when a PnP event of the specified category occurs.
IoRegisterShutdownNotification	The IoRegisterShutdownNotification routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down.
IoReleaseRemoveLock	The IoReleaseRemoveLock routine releases a remove lock acquired with a previous call to IoAcquireRemoveLock.
IoReleaseRemoveLockAndWait	The IoReleaseRemoveLockAndWait routine releases a remove lock that the driver acquired in a previous call to IoAcquireRemoveLock, and waits until all acquisitions of the lock have been released.
IoRemoveLinkShareAccess	The IoRemoveLinkShareAccess routine removes the access and link share-access information for a given open instance of a file object.

TITLE	DESCRIPTION
IoRemoveShareAccess	The IoRemoveShareAccess routine removes the access and share-access information for a given open instance of a file object.
IoReportDetectedDevice	The IoReportDetectedDevice routine reports a non-PnP device to the PnP manager.
IoReportInterruptActive	The IoReportInterruptActive routine informs the operating system that a registered interrupt service routine (ISR) is active and ready to handle interrupt requests.
IoReportInterruptInactive	The IoReportInterruptInactive routine informs the operating system that a registered interrupt service routine (ISR) is inactive and is not expecting interrupt requests.
IoReportResourceForDetection	The IoReportResourceForDetection routine claims hardware resources in the configuration registry for a legacy device.
IoReportRootDevice	The IoReportRootDevice routine reports a device that cannot be detected by a PnP bus driver to the PnP Manager. IoReportRootDevice allows only one device per driver to be created.
IoReportTargetDeviceChange	The IoReportTargetDeviceChange routine notifies the PnP manager that a custom event has occurred on a device.
IoReportTargetDeviceChangeAsynchronous	The IoReportTargetDeviceChangeAsynchronous routine notifies the PnP manager that a custom event has occurred on a device.
IoRequestDeviceEject	The IoRequestDeviceEject routine notifies the PnP manager that the device eject button was pressed.
IoRequestDpc	The IoRequestDpc routine queues a driver-supplied DpcForIsr routine to complete interrupt-driven I/O processing at a lower IRQL.
IoReuseIrp	The IoReuseIrp routine reinitializes an IRP so that it can be reused.
IoSetActivityIdIrp	The IoSetActivityIdIrp routine associates an activity ID with an IRP.
IoSetActivityIdThread	The IoSetActivityIdThread routine associates an activity ID with the current thread. Drivers should use this routine when they are tracing aware and are issuing I/O on a worker thread.
IoSetCancelRoutine	The IoSetCancelRoutine routine sets up a driver-supplied Cancel routine to be called if a given IRP is canceled.
IoSetCompletionRoutine	The IoSetCompletionRoutine routine registers an IoCompletion routine, which will be called when the next-lower-level driver has completed the requested operation for the given IRP.

TITLE	DESCRIPTION
IoSetCompletionRoutineEx	The IoSetCompletionRoutineEx routine registers an IoCompletion routine, which is called when the next-lower-level driver has completed the requested operation for the given IRP.
IoSetDeviceInterfacePropertyData	The IoSetDeviceInterfacePropertyData routine modifies the current value of a device interface property.
IoSetDeviceInterfaceState	The IoSetDeviceInterfaceState routine enables or disables an instance of a previously registered device interface class.
IoSetDevicePropertyData	The IoSetDevicePropertyData routine modifies the current setting for a device property.
IoSetHardErrorOrVerifyDevice	Lower-level drivers call the IoSetHardErrorOrVerifyDevice routine to identify a removable media device that has encountered an error, so that a file system driver can prompt the user to verify that the medium is valid.
IoSetIoPriorityHint	The IoSetIoPriorityHint routine sets the priority hint value for an IRP.
IoSetLinkShareAccess	The IoSetLinkShareAccess routine sets the access rights for link sharing the specified file object.
IoSetMasterIrpStatus	The IoSetMasterIrpStatus routine conditionally replaces the Status value in an IRP with the specified NTSTATUS value.
IoSetNextIrpStackLocation	The IoSetNextIrpStackLocation routine sets the IRP stack location in a driver-allocated IRP to that of the caller.
IoSetShareAccess	The IoSetShareAccess routine sets the access rights for sharing the given file object.
IoSetShareAccessEx	The IoSetShareAccessEx routine sets the access rights for sharing the specified file object.
IoSetStartIoAttributes	The IoSetStartIoAttributes routine sets attributes for the driver's StartIo routine.
IoSetStartIoAttributes	The IoSetStartIoAttributes routine sets attributes for the driver's StartIo routine.
IoSetSystemPartition	The IoSetSystemPartition routine sets the boot partition for the system.
IoSetThreadHardErrorMode	The IoSetThreadHardErrorMode routine enables or disables hard error reporting for the current thread.
IoSizeOfIrp	The IoSizeOfIrp routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.
IoSizeOfIrp	The IoSizeOfIrp routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.

TITLE	DESCRIPTION
IoSizeofWorkItem	The IoSizeofWorkItem routine returns the size, in bytes, of an IO_WORKITEM structure.
IoStartNextPacket	The IoStartNextPacket routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.
IoStartNextPacket	The IoStartNextPacket routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.
IoStartNextPacketByKey	The IoStartNextPacketByKey routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.
IoStartNextPacketByKey	The IoStartNextPacketByKey routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.
IoStartPacket	The IoStartPacket routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.
IoStartPacket	The IoStartPacket routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.
IoStartTimer	The IoStartTimer routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.
IoStartTimer	The IoStartTimer routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.
IoStopTimer	The IoStopTimer routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.
IoStopTimer	The IoStopTimer routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.
IoTransferActivityId	The IoTransferActivityId routine logs an ETW transfer event using the I/O tracing provider on behalf of the caller. This allows a driver to associate two related activity IDs without requiring a specific provider to be enabled.
IoUninitializeWorkItem	The IoUninitializeWorkItem routine uninitializes a work item that was initialized by IoInitializeWorkItem.

TITLE	DESCRIPTION
IoUnregisterBootDriverCallback	The IoUnRegisterBootDriverCallback routine unregisters a previously registered BOOT_DRIVER_CALLBACK_FUNCTION routine.
IoUnregisterContainerNotification	The IoUnregisterContainerNotification routine cancels a container notification registration that was previously created by the IoRegisterContainerNotification routine.
IoUnregisterPlugPlayNotification	This routine is obsolete in Windows 7 and later versions of Windows. For more information, see the following Remarks section.The IoUnregisterPlugPlayNotification routine removes the registration of a driver's callback routine for a PnP event.
IoUnregisterPlugPlayNotificationEx	The IoUnregisterPlugPlayNotificationEx routine cancels the registration of a driver's callback routine for notifications of Plug and Play (PnP) events.
IoUnregisterShutdownNotification	The IoUnregisterShutdownNotification routine removes a registered driver from the shutdown notification queue.
IoUpdateLinkShareAccess	The IoUpdateLinkShareAccess routine updates the share access for the given file object, usually when the file is being opened.
IoUpdateShareAccess	The IoUpdateShareAccess routine updates the share access for the given file object, usually when the file is being opened.
IoValidateDeviceIoControlAccess	For more information, see the WdmlibIoValidateDeviceIoControlAccess function.
IoVerifyPartitionTable	The IoVerifyPartitionTable routine checks the validity of the partition table for a disk.
IoVolumeDeviceToDosName	The IoVolumeDeviceToDosName routine returns the MS-DOS path for a specified device object that represents a file system volume.
IoWithinStackLimits	The IoWithinStackLimits routine determines whether a region of memory is within the stack limit of the current thread.
IoWMIAllocateInstanceIds	The IoWMIAllocateInstanceIds routine allocates one or more instance IDs that are unique to the GUID.
IoWMIDeviceObjectToInstanceName	The IoWMIDeviceObjectToInstanceName routine determines the instance name for the WMI class instance implemented by the driver that is specified by a device object.
IoWMIDeviceObjectToProviderId	The IoWMIDeviceObjectToProviderId routine translates the specified device object into the corresponding WMI Provider ID.
IoWMIExecuteMethod	The IoWMIExecuteMethod routine runs a WMI class method on the specified WMI data block instance.

TITLE	DESCRIPTION
IoWMIHandleToInstanceName	The IoWMIHandleToInstanceName routine determines the instance name for the WMI class instance implemented by the driver that is specified by a file handle.
IoWMIOpenBlock	The IoWMIOpenBlock routine opens the WMI data block object for the specified WMI class.
IoWMIQueryAllData	The IoWMIQueryAllData routine returns all WMI data blocks that implement a given WMI class.
IoWMIQueryAllDataMultiple	The IoWMIQueryAllDataMultiple routine returns all WMI data blocks that implement one of a set of WMI classes.
IoWMIQuerySingleInstance	The IoWMIQuerySingleInstance routine returns the specified instance of a WMI data block.
IoWMIQuerySingleInstanceMultiple	The IoWMIQuerySingleInstanceMultiple routine returns all WMI data block instances that implement the specified WMI classes with the specified instance names.
IoWMIRegistrationControl	The IoWMIRegistrationControl routine registers or unregisters the caller as a WMI data provider for a specified device object.
IoWMISetNotificationCallback	The IoWMISetNotificationCallback routine registers a notification callback for a WMI event.
IoWMISetSingleInstance	The IoWMISetSingleInstance routine sets the values for properties within the data block instance that matches the specified WMI class and instance name.
IoWMISetSingleItem	The IoWMISetSingleItem routine sets a single property in the data block instance that matches the specified WMI class and instance name.
IoWMISuggestInstanceName	The IoWMISuggestInstanceName routine is used to request that WMI suggest a base name which a driver can use to build WMI instance names for the device.
IoWMIWriteEvent	The IoWMIWriteEvent routine delivers a given event to the user-mode WMI components for notification.
IoWriteErrorLogEntry	The IoWriteErrorLogEntry routine queues a given error log packet to the system error logging thread.
IoWriteErrorLogEntry	The IoWriteErrorLogEntry routine queues a given error log packet to the system error logging thread.
IsListEmpty	The IsListEmpty routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.
IsListEmpty	The IsListEmpty routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.

TITLE	DESCRIPTION
KBUGCHECK_CALLBACK_ROUTINE	The BugCheckCallback routine is executed whenever the system issues a bug check.
KBUGCHECK_REASON_CALLBACK_ROUTINE	Driver-implemented callback functions that the system executes when it issues a bug check.
KDEFERRED_ROUTINE	The callback routine performs actions, after an InterruptService returns, of a threaded DPC, The CustomDpc routine finishes the servicing of an I/O operation, after an InterruptService routine returns.The CustomThreadedDpc routine performs the action of a threaded DPC. The system executes this routine when the threaded DPC runs.The CustomTimerDpc routine executes after a timer object's time interval expires.
KeAcquireSpinLock	The KeAcquireSpinLock routine acquires a spin lock so the caller can synchronize access to shared data in a multiprocessor-safe way by raising IRQL.
KeAcquireSpinLockAtDpcLevel	The KeAcquireSpinLockAtDpcLevel routine acquires a spin lock when the caller is already running at IRQL >= DISPATCH_LEVEL.
KeAreAllApcsDisabled	The KeAreAllApcsDisabled routine indicates whether the calling thread is inside a guarded region or running at IRQL >= APC_LEVEL, which disables all APC delivery.
KeAreApcsDisabled	The KeAreApcsDisabled routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.
KeAreApcsDisabled	The KeAreApcsDisabled routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.
KeBugCheck	The KeBugCheck routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.
KeBugCheckEx	The KeBugCheckEx routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.
KeCancelTimer	The KeCancelTimer routine dequeues a timer object before the timer interval, if any was set, expires.
KeClearEvent	The KeClearEvent routine sets an event to a not-signaled state.
KeConvertAuxiliaryCounterToPerformanceCounter	The KeConvertAuxiliaryCounterToPerformanceCounter routine converts the specified auxiliary counter value into a performance counter value.

TITLE	DESCRIPTION
KeConvertPerformanceCounterToAuxiliaryCounter	The KeConvertPerformanceCounterToAuxiliaryCounter routine converts the specified performance counter value into an auxiliary counter value.
KeDelayExecutionThread	The KeDelayExecutionThread routine puts the current thread into an alertable or nonalertable wait state for a specified interval.
KeDeregisterBoundCallback	The KeDeregisterBoundCallback routine deregisters a user-mode bound exception callback registered by KeRegisterBoundCallback.
KeDeregisterBugCheckCallback	The KeDeregisterBugCheckCallback routine removes a callback routine that was registered by KeRegisterBugCheckCallback.
KeDeregisterBugCheckReasonCallback	The KeDeregisterBugCheckReasonCallback routine removes a callback routine that was registered by KeRegisterBugCheckReasonCallback.
KeDeregisterNmiCallback	The KeDeregisterNmiCallback routine deregisters a nonmaskable interrupt (NMI) callback registered by KeRegisterNmiCallback.
KeDeregisterProcessorChangeCallback	The KeDeregisterProcessorChangeCallback routine unregisters a callback function that was previously registered with the operating system by calling the KeRegisterProcessorChangeCallback routine.
KeEnterCriticalRegion	The KeEnterCriticalRegion routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.
KeEnterCriticalRegion	The KeEnterCriticalRegion routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.
KeEnterGuardedRegion	The KeEnterGuardedRegion routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.
KeEnterGuardedRegion	The KeEnterGuardedRegion routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.
KeExpandKernelStackAndCallout	The KeExpandKernelStackAndCallout routine calls a routine with a guaranteed amount of stack space.
KeFlushIoBuffers	The KeFlushIoBuffers routine flushes the memory region described by an MDL from caches of all processors.
KeFlushQueuedDpcs	The KeFlushQueuedDpcs routine returns after all queued DPCs on all processors have executed.
KefReleaseSpinLockFromDpcLevel	The KeReleaseSpinLockFromDpcLevel routine releases an executive spin lock without changing the IRQL.

TITLE	DESCRIPTION
KeGetCurrentIrql	The KeGetCurrentIrql routine returns the current IRQL.
KeGetCurrentNodeNumber	The KeGetCurrentNodeNumber routine gets the NUMA node number for the logical processor that the caller is running on.
KeGetCurrentNodeNumber	The KeGetCurrentNodeNumber routine gets the NUMA node number for the logical processor that the caller is running on.
KeGetCurrentProcessorNumber	The KeGetCurrentProcessorNumber routine returns the system-assigned number of the current processor on which the caller is running.
KeGetCurrentProcessorNumberEx	The KeGetCurrentProcessorNumberEx routine gets the processor number of the logical processor that the caller is running on.
KeGetCurrentProcessorNumberEx	The KeGetCurrentProcessorNumberEx routine gets the processor number of the logical processor that the caller is running on.
KeGetCurrentThread	The KeGetCurrentThread routine identifies the current thread.
KeGetProcessorIndexFromNumber	The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.
KeGetProcessorIndexFromNumber	The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.
KeGetProcessorNumberFromIndex	The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.
KeGetProcessorNumberFromIndex	The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.
KeGetRecommendedSharedDataAlignment	The KeGetRecommendedSharedDataAlignment routine returns the preferred alignment for memory structures that can be accessed by more than one processor.
KeInitializeCrashDumpHeader	The KeInitializeCrashDumpHeader routine supplies the header information the system requires for a crash dump file.
KeInitializeCrashDumpHeader	The KeInitializeCrashDumpHeader routine supplies the header information the system requires for a crash dump file.
KeInitializeDeviceQueue	The KeInitializeDeviceQueue routine initializes a device queue object to a not-busy state.
KeInitializeDpc	The KeInitializeDpc routine initializes a DPC object, and registers a CustomDpc routine for that object.

TITLE	DESCRIPTION
KeInitializeEvent	The KeInitializeEvent routine initializes an event object as a synchronization (single waiter) or notification type event and sets it to a signaled or not-signaled state.
KeInitializeGuardedMutex	The KeInitializeGuardedMutex routine initializes a guarded mutex.
KeInitializeMutex	The KeInitializeMutex routine initializes a mutex object, setting it to a signaled state.
KeInitializeSemaphore	The KeInitializeSemaphore routine initializes a semaphore object with a specified count and specifies an upper limit that the count can attain.
KeInitializeSpinLock	The KeInitializeSpinLock routine initializes a variable of type KSPIN_LOCK.
KeInitializeThreadedDpc	The KeInitializeThreadedDpc routine initializes a threaded DPC object, and registers a CustomThreadedDpc routine for that object.
KeInitializeTimer	The KeInitializeTimer routine initializes a timer object.
KeInitializeTimerEx	The KeInitializeTimerEx routine initializes an extended kernel timer object.
KeInsertByKeyDeviceQueue	The KeInsertByKeyDeviceQueue routine acquires the spin lock for the specified DeviceQueue and queues an entry according to the specified sort-key value if the device queue is set to a busy state.
KeInsertDeviceQueue	The KeInsertDeviceQueue routine acquires the spin lock for the specified device queue object and, if the device queue is set to a busy state, queues the specified entry.
KeInsertQueueDpc	The KeInsertQueueDpc routine queues a DPC for execution.
KeInvalidateAllCaches	The KeInvalidateAllCaches routine flushes all processor caches.
KeIpiGenericCall	The KeIpiGenericCall routine causes the specified routine to run on all processors simultaneously.
KeLeaveCriticalRegion	The KeLeaveCriticalRegion routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to KeEnterCriticalRegion.
KeLeaveCriticalRegion	The KeLeaveCriticalRegion routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to KeEnterCriticalRegion.
KeLeaveGuardedRegion	The KeLeaveGuardedRegion routine exits a guarded region entered by KeEnterGuardedRegion.

TITLE	DESCRIPTION
KeLeaveGuardedRegion	The KeLeaveGuardedRegion routine exits a guarded region entered by KeEnterGuardedRegion.
KeLowerIrql	The KeLowerIrql routine restores the IRQL on the current processor to its original value.
KeMemoryBarrier	The KeMemoryBarrier routine creates a barrier at its position in the code—across which the compiler and the processor cannot move any operations.
KePulseEvent	The KePulseEvent routine atomically sets an event object to a signaled state, attempts to satisfy as many waits as possible, and then resets the event object to a not-signaled state.
KeQueryActiveGroupCount	The KeQueryActiveGroupCount routine returns the number of active processor groups in a multiprocessor system.
KeQueryActiveGroupCount	The KeQueryActiveGroupCount routine returns the number of active processor groups in a multiprocessor system.
KeQueryActiveProcessorCount	The KeQueryActiveProcessorCount routine returns the number of currently active processors.
KeQueryActiveProcessorCount	The KeQueryActiveProcessorCount routine returns the number of currently active processors.
KeQueryActiveProcessorCountEx	The KeQueryActiveProcessorCountEx routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.
KeQueryActiveProcessorCountEx	The KeQueryActiveProcessorCountEx routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.
KeQueryActiveProcessors	The KeQueryActiveProcessors routine returns a bitmask of the currently active processors.
KeQueryActiveProcessors	The KeQueryActiveProcessors routine returns a bitmask of the currently active processors.
KeQueryAuxiliaryCounterFrequency	The KeQueryAuxiliaryCounterFrequency routine returns frequency of the auxiliary counter in units of Hz.
KeQueryDpcWatchdogInformation	The KeQueryDpcWatchdogInformation routine returns the deferred procedure call (DPC) watchdog timer values for the current processor.
KeQueryGroupAffinity	The KeQueryGroupAffinity routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.
KeQueryGroupAffinity	The KeQueryGroupAffinity routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.

TITLE	DESCRIPTION
KeQueryHardwareCounterConfiguration	The KeQueryHardwareCounterConfiguration routine queries the operating system for the list of hardware counters to use for thread profiling.
KeQueryHighestNodeNumber	The KeQueryHighestNodeNumber routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryHighestNodeNumber	The KeQueryHighestNodeNumber routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryInterruptTime	The KeQueryInterruptTime routine returns the current value of the system interrupt time count, with accuracy to within system clock tick.
KeQueryInterruptTimePrecise	The KeQueryInterruptTimePrecise routine returns the current value of the system interrupt time count, with accuracy to within a microsecond.
KeQueryLogicalProcessorRelationship	The KeQueryLogicalProcessorRelationship routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.
KeQueryLogicalProcessorRelationship	The KeQueryLogicalProcessorRelationship routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.
KeQueryMaximumGroupCount	The KeQueryMaximumGroupCount routine returns the maximum number of groups in a multiprocessor system.
KeQueryMaximumGroupCount	The KeQueryMaximumGroupCount routine returns the maximum number of groups in a multiprocessor system.
KeQueryMaximumProcessorCount	The KeQueryMaximumProcessorCount routine returns the maximum number of processors.
KeQueryMaximumProcessorCount	The KeQueryMaximumProcessorCount routine returns the maximum number of processors.
KeQueryMaximumProcessorCountEx	The KeQueryMaximumProcessorCountEx routine returns the maximum number of logical processors in a specified group in a multiprocessor system.
KeQueryMaximumProcessorCountEx	The KeQueryMaximumProcessorCountEx routine returns the maximum number of logical processors in a specified group in a multiprocessor system.
KeQueryNodeActiveAffinity	The KeQueryNodeActiveAffinity routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.

TITLE	DESCRIPTION
KeQueryNodeActiveAffinity	The KeQueryNodeActiveAffinity routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryNodeMaximumProcessorCount	The KeQueryNodeMaximumProcessorCount routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.
KeQueryNodeMaximumProcessorCount	The KeQueryNodeMaximumProcessorCount routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.
KeQueryPerformanceCounter	The KeQueryPerformanceCounter routine retrieves the current value and frequency of the performance counter.Use KeQueryPerformanceCounter to acquire high resolution (<1us) time stamps for time interval measurements.
KeQueryPerformanceCounter	The KeQueryPerformanceCounter routine retrieves the current value and frequency of the performance counter.Use KeQueryPerformanceCounter to acquire high resolution (<1us) time stamps for time interval measurements.
KeQueryPriorityThread	The KeQueryPriorityThread routine returns the current priority of a particular thread.
KeQueryRuntimeThread	The KeQueryRuntimeThread routine reports the accumulated kernel-mode and user-mode run time of a thread, in clock ticks.
KeQuerySystemTime	The KeQuerySystemTime routine obtains the current system time.
KeQuerySystemTimePrecise	The KeQuerySystemTimePrecise routine retrieves the current system time, and is more precise than the KeQuerySystemTime routine.
KeQueryTickCount	The KeQueryTickCount routine maintains a count of the interval timer interrupts that have occurred since the system was booted.
KeQueryTickCount	The KeQueryTickCount routine maintains a count of the interval timer interrupts that have occurred since the system was booted.
KeQueryTimeIncrement	The KeQueryTimeIncrement routine returns the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.
KeQueryTotalCycleTimeThread	The KeQueryTotalCycleTimeThread routine returns the accumulated cycle time for the specified thread.

TITLE	DESCRIPTION
KeQueryUnbiasedInterruptTime	The KeQueryUnbiasedInterruptTime routine returns the current value of the system interrupt time count.
KeRaiseIrql	The KeRaiseIrql routine raises the hardware priority to the specified IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeRaiseIrqlToDpcLevel	The KeRaiseIrqlToDpcLevel routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeRaiseIrqlToDpcLevel	The KeRaiseIrqlToDpcLevel routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeReadStateEvent	The KeReadStateEvent routine returns the current state, signaled or not-signaled, of an event object.
KeReadStateMutex	The KeReadStateMutex routine returns the current state, signaled or not-signaled, of the specified mutex object.
KeReadStateSemaphore	The KeReadStateSemaphore routine returns the current state, signaled or not-signaled, of the specified semaphore object.
KeReadStateTimer	The KeReadStateTimer routine reads the current state of a timer object.
KeRegisterBoundCallback	The KeRegisterBoundCallback routine registers a routine to be called whenever a user-mode bound exception occurs.
KeRegisterBugCheckCallback	The KeRegisterBugCheckCallback routine registers a BugCheckCallback routine, which executes when the operating system issues a bug check.
KeRegisterBugCheckReasonCallback	The KeRegisterBugCheckReasonCallback routine registers a BugCheckDumpIoCallback, BugCheckSecondaryDumpDataCallback, or BugCheckAddPagesCallback routine, which executes when the operating system issues a bug check.
KeRegisterNmiCallback	The KeRegisterNmiCallback routine registers a routine to be called whenever a nonmaskable interrupt (NMI) occurs.
KeRegisterProcessorChangeCallback	The KeRegisterProcessorChangeCallback routine registers a callback function with the operating system so that the operating system will notify the driver when a new processor is added to the hardware partition.
KeReleaseGuardedMutex	The KeReleaseGuardedMutex routine releases a guarded mutex acquired with KeAcquireGuardedMutex or KeTryToAcquireGuardedMutex.
KeReleaseGuardedMutexUnsafe	The KeReleaseGuardedMutexUnsafe routine releases a guarded mutex acquired by KeAcquireGuardedMutexUnsafe.

TITLE	DESCRIPTION
KeReleaseInStackQueuedSpinLock	The KeReleaseInStackQueuedSpinLock routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLock.
KeReleaseInStackQueuedSpinLockForDpc	The KeReleaseInStackQueuedSpinLockForDpc routine releases a queued spin lock that was acquired by calling KeAcquireInStackQueuedSpinLockForDpc.
KeReleaseInStackQueuedSpinLockFromDpcLevel	The KeReleaseInStackQueuedSpinLockFromDpcLevel routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLockAtDpcLevel.
KeReleaseInterruptSpinLock	The KeReleaseInterruptSpinLock routine releases an interrupt spin lock acquired by KeAcquireInterruptSpinLock.
KeReleaseMutex	The KeReleaseMutex routine releases a mutex object, and specifies whether the caller is to call one of the KeWaitXxx routines as soon as KeReleaseMutex returns control.
KeReleaseSemaphore	The KeReleaseSemaphore routine releases the specified semaphore object.
KeReleaseSpinLock	The KeReleaseSpinLock routine releases a spin lock and restores the original IRQL at which the caller was running.
KeReleaseSpinLockForDpc	The KeReleaseSpinLockForDpc routine releases a spin lock that was acquired by calling KeAcquireSpinLockForDpc.
KeReleaseSpinLockFromDpcLevel	The KeReleaseSpinLockFromDpcLevel routine releases an executive spin lock without changing the IRQL.
KeRemoveByKeyDeviceQueue	The KeRemoveByKeyDeviceQueue routine removes an entry, selected according to a sort key value, from the specified device queue.
KeRemoveDeviceQueue	The KeRemoveDeviceQueue routine removes an entry from the head of a specified device queue.
KeRemoveEntryDeviceQueue	The KeRemoveEntryDeviceQueue routine returns whether the specified entry is in the device queue and removes it, if it was queued, from the device queue.
KeRemoveQueueDpc	The KeRemoveQueueDpc routine removes the specified DPC object from the system DPC queue.
KeResetEvent	The KeResetEvent routine resets a specified event object to a not-signaled state and returns the previous state of that event object.
KeRestoreExtendedProcessorState	The KeRestoreExtendedProcessorState routine restores extended processor state information that was previously saved.

TITLE	DESCRIPTION
KeRestoreFloatingPointState	The KeRestoreFloatingPointState routine restores the nonvolatile floating-point context saved by the preceding call to KeSaveFloatingPointState.
KeRevertToUserAffinityThreadEx	The KeRevertToUserAffinityThreadEx routine restores the previous affinity of the current thread.
KeRevertToUserGroupAffinityThread	The KeRevertToUserGroupAffinityThread routine restores the group affinity of the calling thread to its original value at the time that the thread was created.
KeSaveExtendedProcessorState	The KeSaveExtendedProcessorState routine saves extended processor state information.
KeSaveFloatingPointState	The KeSaveFloatingPointState routine saves the nonvolatile floating-point context so the caller can carry out floating-point operations.
KeSetBasePriorityThread	The KeSetBasePriorityThread routine sets the run-time priority, relative to the current process, for a given thread.
KeSetCoalescableTimer	The KeSetCoalescableTimer routine sets the initial expiration time and period of a timer object and specifies how much delay can be tolerated in the expiration times.
KeSetEvent	The KeSetEvent routine sets an event object to a signaled state if the event was not already signaled, and returns the previous state of the event object.
KeSetHardwareCounterConfiguration	The KeSetHardwareCounterConfiguration routine specifies a list of hardware counters to use for thread profiling.
KeSetImportanceDpc	The KeSetImportanceDpc routine specifies how soon the DPC routine is run.
KeSetImportanceDpc	The KeSetImportanceDpc routine specifies how soon the DPC routine is run.
KeSetKernelStackSwapEnable	The KeSetKernelStackSwapEnable routine enables and disables swapping of the caller's stack to disk.
KeSetPriorityThread	The KeSetPriorityThread routine sets the run-time priority of a driver-created thread.
KeSetSystemAffinityThread	The KeSetSystemAffinityThread routine sets the system affinity of the current thread.
KeSetSystemAffinityThreadEx	The KeSetSystemAffinityThreadEx routine sets the system affinity of the current thread.
KeSetSystemGroupAffinityThread	The KeSetSystemGroupAffinityThread routine changes the group number and affinity mask of the calling thread.

TITLE	DESCRIPTION
KeSetTargetProcessorDpc	The KeSetTargetProcessorDpc routine specifies the processor that a DPC routine will be run on.
KeSetTargetProcessorDpc	The KeSetTargetProcessorDpc routine specifies the processor that a DPC routine will be run on.
KeSetTargetProcessorDpcEx	The KeSetTargetProcessorDpcEx routine specifies the processor that a DPC routine will run on.
KeSetTimer	The KeSetTimer routine sets the absolute or relative interval at which a timer object is to be set to a signaled state and, optionally, supplies a CustomTimerDpc routine to be executed when that interval expires.
KeSetTimerEx	The KeSetTimerEx routine sets the absolute or relative interval at which a timer object is to be set to a signaled state, optionally supplies a CustomTimerDpc routine to be executed when that interval expires, and optionally supplies a recurring interval for the timer.
KeStallExecutionProcessor	The KeStallExecutionProcessor routine stalls the caller on the current processor for a specified time interval.
KeStallExecutionProcessor	The KeStallExecutionProcessor routine stalls the caller on the current processor for a specified time interval.
KeSynchronizeExecution	The KeSynchronizeExecution routine synchronizes the execution of the specified routine with the interrupt service routine (ISR) that is assigned to a set of one or more interrupt objects.
KeTestSpinLock	The KeTestSpinLock routine tests for the availability of a spin lock.
KeTryToAcquireSpinLockAtDpcLevel	The KeTryToAcquireSpinLockAtDpcLevel routine attempts to acquire a spin lock at DISPATCH_LEVEL.
KeWaitForMultipleObjects	The KeWaitForMultipleObjects routine puts the current thread into an alertable or nonalertable wait state until any or all of a number of dispatcher objects are set to a signaled state or (optionally) until the wait times out.
KeWaitForSingleObject	The KeWaitForSingleObject routine puts the current thread into a wait state until the given dispatcher object is set to a signaled state or (optionally) until the wait times out.
KIPI_BROADCAST_WORKER	The IpiGenericCall routine runs simultaneously on all processors.
KMESSAGE_SERVICE_ROUTINE	An InterruptMessageService routine services a message-signaled interrupt.
KSERVICE_ROUTINE	The InterruptService routine (ISR) quickly services a device interrupt and schedules post-interrupt processing of received data, if necessary.

TITLE	DESCRIPTION
KSTART_ROUTINE	The ThreadStart routine provides an entry point for a driver-created system thread.
KSYNCHRONIZE_ROUTINE	The SynchCritSection routine is used to access hardware resources or driver data that are shared with a driver's InterruptService routine.
MM_MDL_ROUTINE	A driver-supplied callback routine that is invoked after a memory descriptor list (MDL) is mapped by calling the MmMapMdl function.
MmAdvanceMdl	The MmAdvanceMdl routine advances the beginning of an MDL's virtual memory range by the specified number of bytes.
MmAllocateContiguousMemory	The MmAllocateContiguousMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemory	The MmAllocateContiguousMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCache	The MmAllocateContiguousMemorySpecifyCache routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCache	The MmAllocateContiguousMemorySpecifyCache routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCacheNode	The MmAllocateContiguousMemorySpecifyCacheNode routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCacheNode	The MmAllocateContiguousMemorySpecifyCacheNode routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousNodeMemory	The MmAllocateContiguousNodeMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousNodeMemory	The MmAllocateContiguousNodeMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateMappingAddress	The MmAllocateMappingAddress routine reserves a range of system virtual address space of the specified size.
MmAllocateMdlForIoSpace	The MmAllocateMdlForIoSpace routine allocates an MDL and initializes this MDL to describe a set of physical address ranges in I/O address space.

TITLE	DESCRIPTION
MmAllocateNodePagesForMdlEx	The MmAllocateNodePagesForMdlEx routine allocates nonpaged physical memory from an ideal node, and allocates an MDL structure to describe this memory.
MmAllocateNonCachedMemory	The MmAllocateNonCachedMemory routine allocates a virtual address range of noncached and cache-aligned memory.
MmAllocatePagesForMdl	The MmAllocatePagesForMdl routine allocates zero-filled, nonpaged, physical memory pages to an MDL.
MmAllocatePagesForMdlEx	The MmAllocatePagesForMdlEx routine allocates nonpaged, physical memory pages to an MDL.
MmBuildMdlForNonPagedPool	The MmBuildMdlForNonPagedPool routine receives an MDL that specifies a nonpaged virtual memory buffer, and updates it to describe the underlying physical pages.
MmCopyMemory	The MmCopyMemory routine copies the specified range of virtual or physical memory into the caller-supplied buffer.
MmFreeContiguousMemory	The MmFreeContiguousMemory routine releases a range of physically contiguous memory that was allocated by an MmAllocateContiguousMemoryXxx routine.
MmFreeContiguousMemory	The MmFreeContiguousMemory routine releases a range of physically contiguous memory that was allocated by an MmAllocateContiguousMemoryXxx routine.
MmFreeContiguousMemorySpecifyCache	The MmFreeContiguousMemorySpecifyCache routine frees a buffer that was allocated by an MmAllocateContiguousMemorySpecifyCacheXxx routine.
MmFreeContiguousMemorySpecifyCache	The MmFreeContiguousMemorySpecifyCache routine frees a buffer that was allocated by an MmAllocateContiguousMemorySpecifyCacheXxx routine.
MmFreeMappingAddress	The MmFreeMappingAddress routine frees a range of virtual memory reserved by the MmAllocateMappingAddress routine.
MmFreeNonCachedMemory	The MmFreeNonCachedMemory routine releases a range of noncached memory that was allocated by the MmAllocateNonCachedMemory routine.
MmFreePagesFromMdl	The MmFreePagesFromMdl routine frees all the physical pages that are described by an MDL that was created by the MmAllocatePagesForMdl routine.
MmGetPhysicalAddress	The MmGetPhysicalAddress routine returns the physical address corresponding to a valid nonpaged virtual address.
MmGetSystemAddressForMdl	The MmGetSystemAddressForMdl routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me.

TITLE	DESCRIPTION
MmGetSystemRoutineAddress	The MmGetSystemRoutineAddress routine returns a pointer to a function specified by SystemRoutineName.
MmIsAddressValid	The MmIsAddressValid routine checks whether a page fault will occur for a read or write operation at a given virtual address.Warning We do not recommend using this function.
MmIsDriverSuspectForVerifier	The MmIsDriverSuspectForVerifier routine indicates whether the driver represented by the specified driver object is in the list of drivers that are selected to be verified by Driver Verifier.
MmIsDriverVerifying	The MmIsDriverVerifying routine indicates whether the kernel-mode driver that is identified by the specified driver object is being verified or calls a driver that is being verified by Driver Verifier.
MmIsDriverVerifyingByAddress	The MmIsDriverVerifyingByAddress routine checks whether the kernel-mode driver that is identified by the specified image address is being verified or calls a driver that is being verified by Driver Verifier.
MmIsThisAnNtAsSystem	The MmIsThisAnNtAsSystem routine is obsolete for Windows XP and later versions of Windows. Use RtlGetVersion or RtlVerifyVersionInfo instead.
MmLockPagableCodeSection	The MmLockPagableCodeSection routine locks a section of driver code, containing a set of driver routines marked with a special compiler directive, into system space.
MmLockPagableDataSection	The MmLockPagableDataSection routine locks an entire section of driver data into system space.
MmLockPagableSectionByHandle	The MmLockPagableSectionByHandle routine locks a pageable code or data section into system memory by incrementing the reference count on the handle to the section.
MmMapIoSpace	The MmMapIoSpace routine maps the given physical address range to nonpaged system space.
MmMapIoSpaceEx	The MmMapIoSpaceEx routine maps the given physical address range to non-paged system space using the specified page protection.
MmMapLockedPages	The MmMapLockedPages routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me.
MmMapLockedPagesSpecifyCache	The MmMapLockedPagesSpecifyCache routine maps the physical pages that are described by an MDL to a virtual address, and enables the caller to specify the cache attribute that is used to create the mapping.
MmMapLockedPagesWithReservedMapping	The MmMapLockedPagesWithReservedMapping routine maps all or part of an address range that was previously reserved by the MmAllocateMappingAddress routine.

TITLE	DESCRIPTION
MmMapMdl	This function maps physical pages described by a memory descriptor list (MDL) into the system virtual address space.
MmPageEntireDriver	The MmPageEntireDriver routine causes all of a driver's code and data to be made pageable, overriding the attributes of the various sections that make up the driver's image.
MmProbeAndLockPages	The MmProbeAndLockPages routine probes the specified virtual memory pages, makes them resident, and locks them in memory.
MmProbeAndLockSelectedPages	The MmProbeAndLockSelectedPages routine probes the selected virtual memory pages, makes them resident, and locks them in memory.
MmProtectMdlSystemAddress	The MmProtectMdlSystemAddress routine sets the protection type for a memory address range.
MmQuerySystemSize	The MmQuerySystemSize routine returns an estimate of the amount of memory in the system.
MmResetDriverPaging	The MmResetDriverPaging routine resets the pageable status of a driver's sections to that specified when the driver was compiled.
MmSecureVirtualMemory	The MmSecureVirtualMemory routine secures a user-space memory address range so that it cannot be freed and its protection type cannot be made more restrictive.
MmSizeOfMdl	The MmSizeOfMdl routine returns the number of bytes to allocate for an MDL describing a given address range.
MmUnlockPagableImageSection	The MmUnlockPagableImageSection routine releases a section of driver code or driver data, previously locked into system space with MmLockPagableCodeSection, MmLockPagableDataSection or MmLockPagableSectionByHandle, so the section can be paged out again.
MmUnlockPages	The MmUnlockPages routine unlocks the physical pages that are described by the specified memory descriptor list (MDL).
MmUnmapIoSpace	The MmUnmapIoSpace routine unmaps a specified range of physical addresses previously mapped by MmMapIoSpace.
MmUnmapLockedPages	The MmUnmapLockedPages routine releases a mapping that was set up by a preceding call to the MmMapLockedPages or MmMapLockedPagesSpecifyCache routine.
MmUnmapReservedMapping	The MmUnmapReservedMapping routine unmaps a memory buffer that was mapped by the MmMapLockedPagesWithReservedMapping routine.

TITLE	DESCRIPTION
MmUnsecureVirtualMemory	The MmUnsecureVirtualMemory routine unsecures a memory address range secured by the MmSecureVirtualMemory routine.
NtAllocateVirtualMemory	The ZwAllocateVirtualMemory routine reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process.
NtClose	The ZwClose routine closes an object handle.
NtCommitComplete	The ZwCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction's data.
NtCommitEnlistment	The ZwCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
NtCommitTransaction	The ZwCommitTransaction routine initiates a commit operation for a specified transaction.
NtCreateEnlistment	The ZwCreateEnlistment routine creates a new enlistment object for a transaction.
NtCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
NtCreateResourceManager	The ZwCreateResourceManager routine creates a resource manager object.
NtCreateSection	The ZwCreateSection routine creates a section object.
NtCreateTransaction	The ZwCreateTransaction routine creates a transaction object.
NtCreateTransactionManager	The ZwCreateTransactionManager routine creates a new transaction manager object.
NtDeviceIoControlFile	The ZwDeviceIoControlFile routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.
NtDuplicateToken	The ZwDuplicateToken function creates a handle to a new access token that duplicates an existing token. This function can create either a primary token or an impersonation token.
NtEnumerateTransactionObject	The ZwEnumerateTransactionObject routine enumerates the KTM objects on a computer.
NtFlushBuffersFileEx	The ZwFlushBuffersFileEx routine is called by a file system filter driver to send a flush request for a given file to the file system. An optional flush operation flag can be set to control how file data is written to storage.

TITLE	DESCRIPTION
NtFreeVirtualMemory	The ZwFreeVirtualMemory routine releases, decommits, or both, a region of pages within the virtual address space of a specified process.
NtFsControlFile	The ZwFsControlFile routine sends a control code directly to a specified file system or file system filter driver, causing the corresponding driver to perform the specified action.
NtGetNotificationResourceManager	The ZwGetNotificationResourceManager routine retrieves the next transaction notification from a specified resource manager's notification queue.
NtLockFile	The NtLockFile routine requests a byte-range lock for the specified file.
NtOpenEnlistment	The ZwOpenEnlistment routine obtains a handle to an existing enlistment object.
NtOpenFile	The ZwOpenFile routine opens an existing file, directory, device, or volume.
NtOpenProcess	The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.
NtOpenProcessTokenEx	The ZwOpenProcessTokenEx routine opens the access token associated with a process.
NtOpenResourceManager	The ZwOpenResourceManager routine returns a handle to an existing resource manager object.
NtOpenThreadTokenEx	The ZwOpenThreadTokenEx routine opens the access token associated with a thread.
NtOpenTransaction	The ZwOpenTransaction routine obtains a handle to an existing transaction object.
NtOpenTransactionManager	The ZwOpenTransactionManager routine obtains a handle to an existing transaction manager object.
NtPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.
NtPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.
NtPrepareComplete	The ZwPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
NtPrepareEnlistment	The ZwPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
NtPrePrepareComplete	The ZwPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

TITLE	DESCRIPTION
NtPrePrepareEnlistment	The ZwPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
NtQueryDirectoryFile	The ZwQueryDirectoryFile routine returns various kinds of information about files in the directory specified by a given file handle.
NtQueryInformationEnlistment	The ZwQueryInformationEnlistment routine retrieves information about a specified enlistment object.
NtQueryInformationFile	The ZwQueryInformationFile routine returns various kinds of information about a file object.
NtQueryInformationResourceManager	The ZwQueryInformationResourceManager routine retrieves information about a specified resource manager object.
NtQueryInformationToken	The ZwQueryInformationToken routine retrieves a specified type of information about an access token. The calling process must have appropriate access rights to obtain the information.
NtQueryInformationTransaction	The ZwQueryInformationTransaction routine retrieves information about a specified transaction.
NtQueryInformationTransactionManager	The ZwQueryInformationTransactionManager routine retrieves information about a specified transaction manager object.
NtQueryObject	The ZwQueryObject routine provides information about a supplied object.
NtQueryQuotaInformationFile	The ZwQueryQuotaInformationFile routine retrieves quota entries associated with the volume specified by the FileHandle parameter.
NtQuerySecurityObject	The ZwQuerySecurityObject routine retrieves a copy of an object's security descriptor.
NtQueryVirtualMemory	The ZwQueryVirtualMemory routine determines the state, protection, and type of a region of pages within the virtual address space of the subject process.
NtQueryVolumeInformationFile	The ZwQueryVolumeInformationFile routine retrieves information about the volume associated with a given file, directory, storage device, or volume.
NtReadFile	The ZwReadFile routine reads data from an open file.
NtReadOnlyEnlistment	The ZwReadOnlyEnlistment routine sets a specified enlistment to be read-only.
NtRecoverEnlistment	The ZwRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.

TITLE	DESCRIPTION
NtRecoverResourceManager	The ZwRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
NtRecoverTransactionManager	The ZwRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
NtRenameTransactionManager	The NtRenameTransactionManager routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.
NtRollbackComplete	The ZwRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
NtRollbackEnlistment	The ZwRollbackEnlistment routine rolls back the transaction that is associated with a specified enlistment.
NtRollbackTransaction	The ZwRollbackTransaction routine initiates a rollback operation for a specified transaction.
NtRollforwardTransactionManager	The ZwRollforwardTransactionManager routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.
NtSetInformationEnlistment	The ZwSetInformationEnlistment routine sets information for a specified enlistment object.
NtSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
NtSetInformationResourceManager	The ZwSetInformationResourceManager routine is not used.
NtSetInformationThread	The ZwSetInformationThread routine sets the priority of a thread.
NtSetInformationToken	The ZwSetInformationToken routine modifies information in a specified token. The calling process must have appropriate access rights to set the information.
NtSetInformationTransaction	The ZwSetInformationTransaction routine sets information for a specified transaction.
NtSetInformationTransactionManager	Do not call this routine from kernel-mode code.
NtSetQuotaInformationFile	The ZwSetQuotaInformationFile routine changes quota entries for the volume associated with the FileHandle parameter. All of the quota entries in the specified buffer are applied to the volume.
NtSetSecurityObject	The ZwSetSecurityObject routine sets an object's security state.

TITLE	DESCRIPTION
NtSinglePhaseReject	The ZwSinglePhaseReject routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.
NtUnlockFile	The ZwUnlockFile routine unlocks a byte-range lock in a file.
NtWriteFile	The ZwWriteFile routine writes data to an open file.
ObCloseHandle	The ObCloseHandle routine closes an object handle.
ObDereferenceObject	The ObDereferenceObject routine decrements the given object's reference count and performs retention checks.
ObDereferenceObjectDeferDelete	The ObDereferenceObjectDeferDelete routine decrements the reference count for the given object, checks for object retention, and avoids deadlocks.
ObDereferenceObjectDeferDeleteWithTag	The ObDereferenceObjectDeferDeleteWithTag routine decrements the reference count for the specified object, defers deletion of the object to avoid deadlocks, and writes a four-byte tag value to the object to support object reference tracing.
ObDereferenceObjectWithTag	The ObDereferenceObjectWithTag routine decrements the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.
ObfReferenceObject	The ObReferenceObject routine increments the reference count to the given object.
ObGetObjectSecurity	The ObGetObjectSecurity routine gets the security descriptor for a given object.
ObReferenceObject	The ObReferenceObject routine increments the reference count to the given object.
ObReferenceObjectByHandle	The ObReferenceObjectByHandle routine provides access validation on the object handle, and, if access can be granted, returns the corresponding pointer to the object's body.
ObReferenceObjectByHandleWithTag	The ObReferenceObjectByHandleWithTag routine increments the reference count of the object that is identified by the specified handle, and writes a four-byte tag value to the object to support object reference tracing.
ObReferenceObjectByPointer	The ObReferenceObjectByPointer routine increments the pointer reference count for a given object.
ObReferenceObjectByPointerWithTag	The ObReferenceObjectByPointerWithTag routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.

TITLE	DESCRIPTION
ObReferenceObjectWithTag	The ObReferenceObjectWithTag routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.
ObRegisterCallbacks	The ObRegisterCallbacks routine registers a list of callback routines for thread, process, and desktop handle operations.
ObReleaseObjectSecurity	The ObReleaseObjectSecurity routine is the reciprocal to ObGetObjectSecurity.
ObUnRegisterCallbacks	The ObUnRegisterCallbacks routine unregisters a set of callback routines that were registered with the ObRegisterCallbacks routine.
PALLOCATE_ADAPTER_CHANNEL	The AllocateAdapterChannel routine prepares the system for a DMA operation on behalf of the target device object, and then calls the driver-supplied AdapterControl routine to carry out the DMA operation.
PALLOCATE_ADAPTER_CHANNEL_EX	The AllocateAdapterChannelEx routine allocates the resources that are needed to perform a DMA transfer, and then calls the driver-supplied AdapterControl routine to initiate the DMA transfer.
PALLOCATE_COMMON_BUFFER	The AllocateCommonBuffer routine allocates memory and maps it so that it is simultaneously accessible from both the processor and a device for DMA operations.
PALLOCATE_COMMON_BUFFER_EX	The AllocateCommonBufferEx routine allocates memory for a common buffer and maps this memory so that it can be accessed both by the processor and by a device that performs DMA operations.
PBUILD_MDL_FROM_SCATTER_GATHER_LIST	The BuildMdlFromScatterGatherList routine builds an MDL from a scatter/gather list allocated by the system.Note This routine is reserved for system use.
PBUILD_SCATTER_GATHER_LIST	The BuildScatterGatherList routine prepares the system for a DMA operation, using a driver-supplied buffer to build the scatter/gather list.
PBUILD_SCATTER_GATHER_LIST_EX	The BuildScatterGatherListEx routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.
PCALCULATE_SCATTER_GATHER_LIST_SIZE	The CalculateScatterGatherList routine calculates the size, in bytes, of scatter/gather list necessary to hold a given buffer.
PCANCEL_ADAPTER_CHANNEL	The CancelAdapterChannel routine tries to cancel a pending request to allocate a DMA channel.
PCANCEL_MAPPED_TRANSFER	The CancelMappedTransfer routine cancels a system DMA transfer that is currently mapped to an adapter.

TITLE	DESCRIPTION
PCI_MSIX_MASKUNMASK_ENTRY	The MaskTableEntry routine masks an interrupt in the MSI-X hardware interrupt table.
PCI_MSIX_SET_ENTRY	The SetTableEntry routine sets the message ID for a table entry in the MSI-X hardware interrupt table.
PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK	The ClfsAdvanceTailCallback function advances the base log sequence number (LSN) of the client's log.
PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK	The ClfsLogGrowthCompleteCallback function implements the actions that the client will take when space is freed in a log that had previously been full.
PCLFS_CLIENT_LOG_UNPINNED_CALLBACK	The ClfsLogUnpinnedCallback function implements any actions that the client will take when a log that had previously been pinned becomes unpinned.
PCONFIGURE_ADAPTER_CHANNEL	The ConfigureAdapterChannel routine invokes a custom function that is implemented by the DMA controller represented by an adapter object.
PCREATE_PROCESS_NOTIFY_ROUTINE	Process-creation callback implemented by a driver to track the system-wide creation and deletion of processes against the driver's internal state.
PCREATE_PROCESS_NOTIFY_ROUTINE_EX	A callback routine implemented by a driver to notify the caller when a process is created or exits.
PCREATE_THREAD_NOTIFY_ROUTINE	A callback routine implemented by a driver to notify the caller when a thread is created or deleted.
PDEVICE_RESET_HANDLER	The DeviceReset routine is used to reset and recover a malfunctioning device.
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

TITLE	DESCRIPTION
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE	The PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE	The PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE	The PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE	The PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

TITLE	DESCRIPTION
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.
PEPCALLBACKNOTIFYACPI	An AcceptAcpiNotification event callback routine handles ACPI notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYACPI	An AcceptAcpiNotification event callback routine handles ACPI notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYDPM	An AcceptDeviceNotification event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYDPM	An AcceptDeviceNotification event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYPPM	An AcceptProcessorNotification event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYPPM	An AcceptProcessorNotification event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKPOWERONCRASHDUMPDEVICE	The PowerOnDumpDeviceCallback callback routine turns on the crash-dump device.
PEPCALLBACKPOWERONCRASHDUMPDEVICE	The PowerOnDumpDeviceCallback callback routine turns on the crash-dump device.
PFLUSH_ADAPTER_BUFFERS	The FlushAdapterBuffers routine flushes any data remaining in the system DMA controller's internal cache or in a bus-master adapter's internal cache at the end of a DMA transfer operation.
PFLUSH_ADAPTER_BUFFERS_EX	The FlushAdapterBuffersEx routine flushes any data that remains in the data cache at the end of a DMA transfer operation performed by a system DMA controller or bus-master device.
PFREE_ADAPTER_CHANNEL	The FreeAdapterChannel routine releases the system DMA controller when a driver has completed all DMA operations necessary to satisfy the current IRP.
PFREE_ADAPTER_OBJECT	The FreeAdapterObject routine releases the specified adapter object after a driver has completed all DMA operations.
PFREE_COMMON_BUFFER	The FreeCommonBuffer routine frees a common buffer allocated by AllocateCommonBuffer, along with all resources the buffer uses.

TITLE	DESCRIPTION
PFREE_MAP_REGISTERS	The FreeMapRegisters routine releases a set of map registers that were saved from a call to AllocateAdapterChannel.
PGET_DMA_ADAPTER_INFO	The GetDmaAdapterInfo routine retrieves information about the hardware capabilities of a system DMA channel.
PGET_DMA_ALIGNMENT	The GetDmaAlignment routine returns the alignment requirements of the DMA system.
PGET_DMA_TRANSFER_INFO	The GetDmaTransferInfo routine calculates the allocation requirements for a scatter/gather DMA transfer.
PGET_LOCATION_STRING	The PnpGetLocationString routine provides the device-specific part of the device's SPDRP_LOCATION_PATHS property.
PGET_SCATTER_GATHER_LIST	The GetScatterGatherList routine prepares the system for a DMA scatter/gather operation on behalf of the target device object, through either the system DMA controller or a bus-master adapter.
PGET_SCATTER_GATHER_LIST_EX	The GetScatterGatherListEx routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.
pHalExamineMBR	The HalExamineMBR routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.
PINITIALIZE_DMA_TRANSFER_CONTEXT	The InitializeDmaTransferContext routine initializes an opaque DMA transfer context that is used to track pending allocations of DMA resources.
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.
PLOAD_IMAGE_NOTIFY_ROUTINE	Called by the operating system to notify the driver when a driver image or a user image (for example, a DLL or EXE) is mapped into virtual memory.

TITLE	DESCRIPTION
PMAP_TRANSFER	The MapTransfer routine sets up map registers for an adapter object to map a DMA transfer from a locked-down buffer.
PMAP_TRANSFER_EX	The MapTransferEx routine sets up map registers to map the physical addresses in a scatter/gather list to the logical addresses that are required to do a DMA transfer.
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK	The ComponentActiveConditionCallback callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK	The ComponentActiveConditionCallback callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK	The ComponentCriticalTransitionCallback callback routine handles a transition of the specified component between the F0 (fully on) and low-power Fx component power states.
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK	The ComponentCriticalTransitionCallback callback routine handles a transition of the specified component between the F0 (fully on) and low-power Fx component power states.
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK	The ComponentIdleConditionCallback callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK	The ComponentIdleConditionCallback callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
PO_FX_COMPONENT_IDLE_STATE_CALLBACK	The ComponentIdleStateCallback callback routine notifies the driver of a pending change to the Fx power state of the specified component.
PO_FX_COMPONENT_IDLE_STATE_CALLBACK	The ComponentIdleStateCallback callback routine notifies the driver of a pending change to the Fx power state of the specified component.
PO_FX_COMPONENT_PERF_STATE_CALLBACK	The ComponentPerfStateCallback callback routine notifies the driver that its request to change the performance state of a component is complete.
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK	The DevicePowerNotRequiredCallback callback routine notifies the device driver that the device is not required to stay in the D0 power state.
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK	The DevicePowerNotRequiredCallback callback routine notifies the device driver that the device is not required to stay in the D0 power state.
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK	The DevicePowerRequiredCallback callback routine notifies the device driver that the device must enter and remain in the D0 power state.

TITLE	DESCRIPTION
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK	The DevicePowerRequiredCallback callback routine notifies the device driver that the device must enter and remain in the D0 power state.
PO_FX_POWER_CONTROL_CALLBACK	The PowerControlCallback callback routine performs a power control operation that is requested by the power management framework (PoFx).
PO_FX_POWER_CONTROL_CALLBACK	The PowerControlCallback callback routine performs a power control operation that is requested by the power management framework (PoFx).
POB_POST_OPERATION_CALLBACK	The ObjectPostCallback routine is called by the operating system after a process or thread handle operation occurs.
POB_PRE_OPERATION_CALLBACK	The ObjectPreCallback routine is called by the operating system when a process or thread handle operation occurs.
PoCallDriver	The PoCallDriver routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoCallDriver	The PoCallDriver routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoClearPowerRequest	The PoClearPowerRequest routine decrements the count for the specified power request type.
PoClearPowerRequest	The PoClearPowerRequest routine decrements the count for the specified power request type.
PoCreatePowerRequest	The PoCreatePowerRequest routine creates a power request object.
PoCreatePowerRequest	The PoCreatePowerRequest routine creates a power request object.
PoDeletePowerRequest	The PoDeletePowerRequest routine deletes a power request object.
PoDeletePowerRequest	The PoDeletePowerRequest routine deletes a power request object.
PoEndDeviceBusy	The PoEndDeviceBusy routine marks the end of a period of time in which the device is busy.
PoEndDeviceBusy	The PoEndDeviceBusy routine marks the end of a period of time in which the device is busy.
PoFxActivateComponent	The PoFxActivateComponent routine increments the activation reference count on the specified component.

TITLE	DESCRIPTION
POFXCALLBACKCRITICALRESOURCE	The TransitionCriticalResource routine changes the active/inactive state of a core system component.
POFXCALLBACKCRITICALRESOURCE	The TransitionCriticalResource routine changes the active/inactive state of a core system component.
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS	The EnumerateUnmaskedInterrupts routine enumerates interrupt sources whose interrupts are unmasked and enabled.
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS	The EnumerateUnmaskedInterrupts routine enumerates interrupt sources whose interrupts are unmasked and enabled.
POFXCALLBACKPLATFORMIDLEVETO	The PlatformIdleVeto routine increments or decrements the veto count for a veto code for a platform idle state.
POFXCALLBACKPLATFORMIDLEVETO	The PlatformIdleVeto routine increments or decrements the veto count for a veto code for a platform idle state.
POFXCALLBACKPROCESSORHALT	The ProcessorHalt routine prepares the processor to be halted.
POFXCALLBACKPROCESSORHALT	The ProcessorHalt routine prepares the processor to be halted.
POFXCALLBACKPROCESSORIDLEVETO	The ProcessorIdleVeto routine increments or decrements the veto count for a veto code for a processor idle state.
POFXCALLBACKPROCESSORIDLEVETO	The ProcessorIdleVeto routine increments or decrements the veto count for a veto code for a processor idle state.
POFXCALLBACKREQUESTCOMMON	The RequestCommon routine is a generic request handler.
POFXCALLBACKREQUESTCOMMON	The RequestCommon routine is a generic request handler.
POFXCALLBACKREQUESTINTERRUPT	The RequestInterrupt routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.
POFXCALLBACKREQUESTINTERRUPT	The RequestInterrupt routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.
POFXCALLBACKREQUESTWORKER	The RequestWorker routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.
POFXCALLBACKREQUESTWORKER	The RequestWorker routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.

TITLE	DESCRIPTION
POFXCALLBACKUPDATEPLATFORMIDLESTATE	The UpdatePlatformIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.
POFXCALLBACKUPDATEPLATFORMIDLESTATE	The UpdatePlatformIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.
POFXCALLBACKUPDATEPROCESSORIDLESTATE	The UpdateProcessorIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.
POFXCALLBACKUPDATEPROCESSORIDLESTATE	The UpdateProcessorIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.
PoFxCompleteDevicePowerNotRequired	The PoFxCompleteDevicePowerNotRequired routine notifies the power management framework (PoFx) that the calling driver has completed its response to a call to the driver's DevicePowerNotRequiredCallback callback routine.
PoFxCompleteIdleCondition	The PoFxCompleteIdleCondition routine informs the power management framework (PoFx) that the specified component has completed a pending change to the idle condition.
PoFxCompleteIdleState	The PoFxCompleteIdleState routine informs the power management framework (PoFx) that the specified component has completed a pending change to an Fx state.
PoFxIdleComponent	The PoFxIdleComponent routine decrements the activation reference count on the specified component.
PoFxIssueComponentPerfStateChange	The PoFxIssueComponentPerfStateChange routine submits a request to place a device component in a particular performance state.
PoFxIssueComponentPerfStateChangeMultiple	The PoFxIssueComponentPerfStateChangeMultiple routine submits a request to change the performance states in multiple performance state sets simultaneously for a device component.
PoFxNotifySurprisePowerOn	The PoFxNotifySurprisePowerOn routine notifies the power management framework (PoFx) that a device was turned on as a side effect of supplying power to some other device.
PoFxPowerControl	The PoFxPowerControl routine sends a power control request to the power management framework (PoFx).
PoFxPowerOnCrashdumpDevice	The PoFxPowerOnCrashdumpDevice routine requests that a crash-dump device be turned on.
PoFxQueryCurrentComponentPerfState	The PoFxQueryCurrentComponentPerfState routine retrieves the active performance state in a component's performance state set.

TITLE	DESCRIPTION
PoFxRegisterComponentPerfStates	The PoFxRegisterComponentPerfStates routine registers a device component for performance state management by the power management framework (PoFx).
PoFxRegisterCoreDevice	The PoFxRegisterCoreDevice routine registers a new core system resource with the Windows power management framework (PoFx).
PoFxRegisterCoreDevice	The PoFxRegisterCoreDevice routine registers a new core system resource with the Windows power management framework (PoFx).
PoFxRegisterCrashdumpDevice	The PoFxRegisterCrashdumpDevice routine registers a crash-dump device.
PoFxRegisterDevice	The PoFxRegisterDevice routine registers a device with the power management framework (PoFx).
PoFxRegisterPlugin	The PoFxRegisterPlugin routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPlugin	The PoFxRegisterPlugin routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPluginEx	The PoFxRegisterPluginEx routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPluginEx	The PoFxRegisterPluginEx routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxReportDevicePoweredOn	The PoFxReportDevicePoweredOn routine notifies the power management framework (PoFx) that the device completed the requested transition to the D0 (fully on) power state.
PoFxSetComponentLatency	The PoFxSetComponentLatency routine specifies the maximum latency that can be tolerated in the transition from the idle condition to the active condition in the specified component.
PoFxSetComponentResidency	The PoFxSetComponentResidency routine sets the estimated time for how long a component is likely to remain idle after the component enters the idle condition.
PoFxSetComponentWake	The PoFxSetComponentWake routine indicates whether the driver arms the specified component to wake whenever the component enters the idle condition.

TITLE	DESCRIPTION
PoFxSetDeviceIdleTimeout	The PoFxSetDeviceIdleTimeout routine specifies the minimum time interval from when the last component of the device enters the idle condition to when the power management framework (PoFx) calls the driver's DevicePowerNotRequiredCallback routine.
PoFxSetTargetDripsDevicePowerState	This routine is called to notify the power manager of the device's target device power state for DRIPS. The driver can override the DRIPS constraint provided by the PEP.
PoFxStartDevicePowerManagement	The PoFxStartDevicePowerManagement routine completes the registration of a device with the power management framework (PoFx) and starts device power management.
PoFxUnregisterDevice	The PoFxUnregisterDevice routine removes the registration of a device from the power management framework (PoFx).
PoGetSystemWake	The PoGetSystemWake routine determines whether a specified IRP has been marked as waking the system from a sleeping state.
PopEntryList	The PopEntryList routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.
PopEntryList	The PopEntryList routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.
PoQueryWatchdogTime	The PoQueryWatchdogTime routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.
PoQueryWatchdogTime	The PoQueryWatchdogTime routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.
PoRegisterDeviceForIdleDetection	The PoRegisterDeviceForIdleDetection routine enables or cancels idle detection and sets idle time-out values for a device.
PoRegisterDeviceForIdleDetection	The PoRegisterDeviceForIdleDetection routine enables or cancels idle detection and sets idle time-out values for a device.
PoRegisterPowerSettingCallback	The PoRegisterPowerSettingCallback routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.
PoRegisterPowerSettingCallback	The PoRegisterPowerSettingCallback routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.
PoRegisterSystemState	The PoRegisterSystemState routine registers the system as busy due to certain activity.

TITLE	DESCRIPTION
PoRegisterSystemState	The PoRegisterSystemState routine registers the system as busy due to certain activity.
PoRequestPowerIrp	The PoRequestPowerIrp routine allocates a power IRP and sends it to the top driver in the device stack for the specified device.
PoSetDeviceBusyEx	The PoSetDeviceBusyEx routine notifies the power manager that the device associated with the specified idle counter is busy.
PoSetDeviceBusyEx	The PoSetDeviceBusyEx routine notifies the power manager that the device associated with the specified idle counter is busy.
PoSetPowerRequest	The PoSetPowerRequest routine increments the count for the specified power request type.
PoSetPowerRequest	The PoSetPowerRequest routine increments the count for the specified power request type.
PoSetPowerState	The PoSetPowerState routine notifies the system of a change in the device power state for a device.
PoSetPowerState	The PoSetPowerState routine notifies the system of a change in the device power state for a device.
PoSetSystemState	Drivers call the PoSetSystemState routine to indicate that the system is active.
PoSetSystemWake	The PoSetSystemWake routine marks the specified IRP as one that contributed to waking the system from a sleep state.
PoStartDeviceBusy	The PoStartDeviceBusy routine marks the start of a period of time in which the device is busy.
PoStartDeviceBusy	The PoStartDeviceBusy routine marks the start of a period of time in which the device is busy.
PoStartNextPowerIrp	The PoStartNextPowerIrp routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoStartNextPowerIrp	The PoStartNextPowerIrp routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoUnregisterPowerSettingCallback	The PoUnregisterPowerSettingCallback routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.

TITLE	DESCRIPTION
PoUnregisterPowerSettingCallback	The PoUnregisterPowerSettingCallback routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.
PoUnregisterSystemState	The PoUnregisterSystemState routine cancels a system state registration created by PoRegisterSystemState.
PoUnregisterSystemState	The PoUnregisterSystemState routine cancels a system state registration created by PoRegisterSystemState.
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK	An EnumerateInterruptSource callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK	An EnumerateInterruptSource callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.
PPUT_DMA_ADAPTER	The PutDmaAdapter routine frees a DMA_ADAPTER structure previously allocated by IoGetDmaAdapter.
PPUT_SCATTER_GATHER_LIST	The PutScatterGatherList routine frees the previously allocated map registers and scatter/gather list used in scatter/gather DMA.
PREAD_DMA_COUNTER	The ReadDmaCounter routine returns the number of bytes remaining to be transferred during the current subordinate DMA operation.
PREENUMERATE_SELF	A ReenumerateSelf routine requests that a bus driver reenumerate a child device.
ProbeForRead	The ProbeForRead routine checks that a user-mode buffer actually resides in the user portion of the address space, and is correctly aligned.
ProbeForWrite	The ProbeForWrite routine checks that a user-mode buffer actually resides in the user-mode portion of the address space, is writable, and is correctly aligned.
PROCESSOR_HALT_ROUTINE	A Halt callback routine transitions the processor to an idle state.
PsAllocSiloContextSlot	This routine allocates a slot that can be used to insert, retrieve, and delete an object in all silos. .
PsAttachSiloToCurrentThread	This routine places a thread temporarily into the specified Silo.
PsCreateSiloContext	This routine creates an object that will be inserted in a Silo.
PsCreateSystemThread	The PsCreateSystemThread routine creates a system thread that executes in kernel mode and returns a handle for the thread.

TITLE	DESCRIPTION
PsDereferenceSiloContext	This routine decrements the reference count on the object.
PsDetachSiloFromCurrentThread	This routine removes a thread from a silo which was added by an attach. For more info about attaching, see the PsAttachSiloToCurrentThread routine.
PsFreeSiloContextSlot	This routine frees the specified slot and makes it available in the system. It undoes the effects of the PsAllocSiloContextSlot routine.
PsGetCurrentProcessId	The PsGetCurrentProcessId routine identifies the current thread's process.
PsGetCurrentServerSilo	This routine returns the effective server silo for the thread.
PsGetCurrentSilo	This routine returns the current silo for the calling thread. First the thread is checked to see if it has been attached to a silo. If not, then the thread is checked to see if it is in a silo.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetCurrentThreadId	The PsGetCurrentThreadId routine identifies the current thread.
PsGetCurrentThreadTeb	The PsGetCurrentThreadTeb routine returns the Thread Environment Block (TEB) of the current thread. The call must be made in kernel-mode.
PsGetEffectiveServerSilo	This routine traverses the parent chain of the Silo until finding the effective server silo or host silo.
PsGetHostSilo	This routine returns the host silo.
PsGetJobServerSilo	This routine returns the effective ServerSilo for the job. The returned pointer is valid as long as the supplied Job object remains referenced.
PsGetJobSilo	This routine returns the first job in the hierarchy that is a Silo. The returned pointer is valid as long as the supplied Job object remains referenced.
PsGetParentSilo	Retrieves the most immediate parent silo in the hierarchy for a given job object.
PsGetPermanentSiloContext	This routine retrieves an object that was inserted in the Silo without incrementing the reference count.
PsGetProcessCreateTimeQuadPart	The PsGetProcessCreateTimeQuadPart routine returns a LONGLONG value that represents the time at which the process was created.

TITLE	DESCRIPTION
PsGetProcessId	The PsGetProcessId routine returns the process identifier (process ID) that is associated with a specified process.
PsGetServerSiloActiveConsoleId	Gets the active console for the current server silo context for the supplied thread.
PsGetSiloContext	This routine retrieves the silo context from the specified silo and slot.
PsGetSiloMonitorContextSlot	This routine returns the silo context slot that was allocated by the monitor during the registration.
PsGetVersion	This function is obsolete in Windows XP and later versions of the Windows operating system. Use RtlGetVersion instead.PsGetVersion returns caller-selected information about the current version of the NT-based operating system.
PsInsertPermanentSiloContext	This routine inserts an object in an empty slot in a Silo.
PsInsertSiloContext	This routine inserts an object in an empty slot in a Silo.
PsIsHostSilo	This routine will check if the supplied Silo is the host silo.
PsIsSystemThread	The PsIsSystemThread routine checks whether a given thread is a system thread.
PsMakeSiloContextPermanent	This routine makes the slot in a silo instance read-only, allowing the object in the slot to be retrieved without affecting the reference count on that object.
PsQueryTotalCycleTimeProcess	The PsQueryTotalCycleTimeProcess routine returns the accumulated cycle time for the specified process.
PsReferenceSiloContext	This routine increments the reference count on the object.
PsRegisterSiloMonitor	This routine registers a server silo monitor that can receive notifications about server silo events.
PsRemoveCreateThreadNotifyRoutine	The PsRemoveCreateThreadNotifyRoutine routine removes a callback routine that was registered by the PsSetCreateThreadNotifyRoutine routine.
PsRemoveLoadImageNotifyRoutine	The PsRemoveLoadImageNotifyRoutine routine removes a callback routine that was registered by the PsSetLoadImageNotifyRoutine routine.
PsRemoveSiloContext	This routine removes an object that was inserted in the Silo.
PsReplaceSiloContext	This routine inserts an object in a Silo.
PsSetCreateProcessNotifyRoutine	The PsSetCreateProcessNotifyRoutine routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

TITLE	DESCRIPTION
PsSetCreateProcessNotifyRoutineEx	The PsSetCreateProcessNotifyRoutineEx routine registers or removes a callback routine that notifies the caller when a process is created or exits.
PsSetCreateProcessNotifyRoutineEx2	The PsSetCreateProcessNotifyRoutineEx2 routine registers or removes a callback routine that notifies the caller when a process is created or deleted.
PsSetCreateThreadNotifyRoutine	The PsSetCreateThreadNotifyRoutine routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.
PsSetCreateThreadNotifyRoutineEx	The PsSetCreateThreadNotifyRoutineEx routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.
PsSetLoadImageNotifyRoutine	The PsSetLoadImageNotifyRoutine routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
PsSetLoadImageNotifyRoutineEx	The PsSetLoadImageNotifyRoutineEx routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
PsStartSiloMonitor	This routine tries to start the server silo monitor.
PsTerminateServerSilo	This routine terminates the specified silo.
PsTerminateSystemThread	The PsTerminateSystemThread routine terminates the current system thread.
PsUnregisterSiloMonitor	This routine unregisters a server silo monitor.
PTM_RM_NOTIFICATION	A resource manager's ResourceManagerNotification callback routine receives and handles transaction notifications.
PushEntryList	The PushEntryList routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
PushEntryList	The PushEntryList routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
READ_PORT_BUFFER_UCHAR	The READ_PORT_BUFFER_UCHAR routine reads a number of bytes from the specified port address into a buffer.
READ_PORT_BUFFER_UCHAR	The READ_PORT_BUFFER_UCHAR routine reads a number of bytes from the specified port address into a buffer.
READ_PORT_BUFFER_ULONG	The READ_PORT_BUFFER_ULONG routine reads a number of ULONG values from the specified port address into a buffer.
READ_PORT_BUFFER_ULONG	The READ_PORT_BUFFER_ULONG routine reads a number of ULONG values from the specified port address into a buffer.

TITLE	DESCRIPTION
READ_PORT_BUFFER_USHORT	The READ_PORT_BUFFER_USHORT routine reads a number of USHORT values from the specified port address into a buffer.
READ_PORT_BUFFER_USHORT	The READ_PORT_BUFFER_USHORT routine reads a number of USHORT values from the specified port address into a buffer.
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.

TITLE	DESCRIPTION
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
RemoveEntryList	The RemoveEntryList routine removes an entry from a doubly linked list of LIST_ENTRY structures.
RemoveEntryList	The RemoveEntryList routine removes an entry from a doubly linked list of LIST_ENTRY structures.

TITLE	DESCRIPTION
RemoveHeadList	The RemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
RemoveHeadList	The RemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
RemoveTailList	The RemoveTailList routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.
RemoveTailList	The RemoveTailList routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.
REQUEST_POWER_COMPLETE	The PowerCompletion callback routine completes the processing of a power IRP.
RTL_QUERY_REGISTRY_ROUTINE	The QueryRoutine routine provides information about a registry value that was requested in a preceding call to the RtlQueryRegistryValues routine.
RTL_RUN_ONCE_INIT_FN	The RunOnceInitialization routine performs a one-time initialization operation.
RtlAnsiStringToUnicodeSize	The RtlAnsiStringToUnicodeSize routine returns the number of bytes required to hold an ANSI string converted into a Unicode string.
RtlAnsiStringToUnicodeString	RtlAnsiStringToUnicodeString converts the given ANSI source string into a Unicode string.
RtlAppendUnicodeStringToString	The RtlAppendUnicodeStringToString routine concatenates two Unicode strings.
RtlAppendUnicodeStringToString	The RtlAppendUnicodeStringToString routine concatenates two Unicode strings.
RtlAppendUnicodeToString	The RtlAppendUnicodeToString routine concatenates the supplied Unicode string to a buffered Unicode string.
RtlAppendUnicodeToString	The RtlAppendUnicodeToString routine concatenates the supplied Unicode string to a buffered Unicode string.
RtlAreBitsClear	The RtlAreBitsClear routine determines whether a given range of bits within a bitmap variable is clear.
RtlAreBitsSet	The RtlAreBitsSet routine determines whether a given range of bits within a bitmap variable is set.
RtlByteToChar	Converts a value of type BYTE to a value of type CHAR.
RtlByteToInt8	Converts a value of type BYTE to a value of type INT8.
RtlCharToInteger	The RtlCharToInteger routine converts a single-byte character string to an integer value in the specified base.

TITLE	DESCRIPTION
RtlCheckBit	The RtlCheckBit routine determines whether a particular bit in a given bitmap variable is clear or set.
RtlCheckRegistryKey	The RtlCheckRegistryKey routine checks for the existence of a given named key in the registry.
RtlClearAllBits	The RtlClearAllBits routine sets all bits in a given bitmap variable to zero.
RtlClearBit	The RtlClearBit routine sets the specified bit in a bitmap to zero.
RtlClearBits	The RtlClearBits routine sets all bits in the specified range of bits in the bitmap to zero.
RtlCmDecodeMemIoResource	The RtlCmDecodeMemIoResource routine provides the starting address and length of a CM_PARTIAL_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.
RtlCmEncodeMemIoResource	The RtlCmEncodeMemIoResource routine updates a CM_PARTIAL_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.
RtlCompareMemory	The RtlCompareMemory routine compares two blocks of memory and returns the number of bytes that match.
RtlCompareString	The RtlCompareString routine compares two counted strings.
RtlCompareUnicodeString	The RtlCompareUnicodeString routine compares two Unicode strings.
RtlCompareUnicodeString	The RtlCompareUnicodeString routine compares two Unicode strings.
RtlConvertLongToLargeInteger	The RtlConvertLongToLargeInteger routine converts the input signed integer to a signed large integer.
RtlConvertLongToLuid	The RtlConvertLongToLuid routine converts a long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.
RtlConvertUlongToLargeInteger	The RtlConvertUlongToLargeInteger routine converts the input unsigned integer to a signed large integer. For Windows XP and later versions of Windows, do not use this routine; use the native support for __int64.
RtlConvertUlongToLuid	The RtlConvertUlongToLuid routine converts an unsigned long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.
RtlCopyMemory	The RtlCopyMemory routine copies the contents of a source memory block to a destination memory block.

TITLE	DESCRIPTION
RtlCopyString	The RtlCopyString routine copies a source string to a destination string.
RtlCopyUnicodeString	The RtlCopyUnicodeString routine copies a source string to a destination string.
RtlCopyUnicodeString	The RtlCopyUnicodeString routine copies a source string to a destination string.
RtlCreateRegistryKey	The RtlCreateRegistryKey routine adds a key object in the registry along a given relative path.
RtlCreateSecurityDescriptor	The RtlCreateSecurityDescriptor routine initializes a new absolute-format security descriptor. On return, the security descriptor is initialized with no system ACL, no discretionary ACL, no owner, no primary group, and all control flags set to zero.
RtlDeleteRegistryValue	The RtlDeleteRegistryValue routine removes the specified entry name and the associated values from the registry along the given relative path.
RtlDowncaseUnicodeChar	The RtlDowncaseUnicodeChar routine converts the specified Unicode character to lowercase.
RtlDWordPtrAdd	Adds two values of type DWORD_PTR.
RtlDWordPtrMult	Multiplies one value of type DWORD_PTR by another.
RtlDWordPtrSub	Subtracts one value of type DWORD_PTR from another.
RtlEqualMemory	The RtlEqualMemory routine compares two blocks of memory to determine whether the specified number of bytes are identical.
RtlEqualString	The RtlEqualString routine compares two counted strings to determine whether they are equal.
RtlEqualUnicodeString	The RtlEqualUnicodeString routine compares two Unicode strings to determine whether they are equal.
RtlEqualUnicodeString	The RtlEqualUnicodeString routine compares two Unicode strings to determine whether they are equal.
RtlExtendCorrelationVector	This routine extends the supplied correlation vector. For a correlation vector of the form X.i, the extended value is X.i.0.
RtlFillMemory	The RtlFillMemory routine fills a block of memory with the specified fill value.
RtlFindClearBits	The RtlFindClearBits routine searches for a range of clear bits of a requested size within a bitmap.

TITLE	DESCRIPTION
RtlFindClearBitsAndSet	The RtlFindClearBitsAndSet routine searches for a range of clear bits of a requested size within a bitmap and sets all bits in the range when it has been located.
RtlFindClearRuns	The RtlFindClearRuns routine finds the specified number of runs of clear bits within a given bitmap.
RtlFindFirstRunClear	The RtlFindFirstRunClear routine searches for the initial contiguous range of clear bits within a given bitmap.
RtlFindLastBackwardRunClear	The RtlFindLastBackwardRunClear routine searches a given bitmap for the preceding clear run of bits, starting from the specified index position.
RtlFindLeastSignificantBit	The RtlFindLeastSignificantBit routine returns the zero-based position of the least significant nonzero bit in its parameter.
RtlFindLongestRunClear	The RtlFindLongestRunClear routine searches for the largest contiguous range of clear bits within a given bitmap.
RtlFindMostSignificantBit	The RtlFindMostSignificantBit routine returns the zero-based position of the most significant nonzero bit in its parameter.
RtlFindNextForwardRunClear	The RtlFindNextForwardRunClear routine searches a given bitmap variable for the next clear run of bits, starting from the specified index position.
RtlFindSetBits	The RtlFindSetBits routine searches for a range of set bits of a requested size within a bitmap.
RtlFindSetBitsAndClear	The RtlFindSetBitsAndClear routine searches for a range of set bits of a requested size within a bitmap and clears all bits in the range when it has been located.
RtlFreeAnsiString	The RtlFreeAnsiString routine releases storage that was allocated by RtlUnicodeStringToAnsiString.
RtlFreeUnicodeString	The RtlFreeUnicodeString routine releases storage that was allocated by RtlAnsiStringToUnicodeString or RtlUpcaseUnicodeString.
RtlFreeUnicodeString	The RtlFreeUnicodeString routine releases storage that was allocated by RtlAnsiStringToUnicodeString or RtlUpcaseUnicodeString.
RtlGetEnabledExtendedFeatures	The RtlGetEnabledExtendedFeatures routine returns a mask of extended processor features that are enabled by the system.
RtlGetVersion	The RtlGetVersion routine returns version information about the currently running operating system.
RtlGUIDFromString	The RtlGUIDFromString routine converts the given Unicode string to a GUID in binary format.

TITLE	DESCRIPTION
RtlHashUnicodeString	The RtlHashUnicodeString routine creates a hash value from a given Unicode string and hash algorithm.
RtlIncrementCorrelationVector	Increments the specified correlation vector. For a correlation vector of the form X.i, the incremented value is be X.(i+1).
RtlInitAnsiString	The RtlInitAnsiString routine initializes a counted string of ANSI characters.
RtlInitializeBitMap	The RtlInitializeBitMap routine initializes the header of a bitmap variable.
RtlInitializeCorrelationVector	Initializes the specified correlation vector with the supplied GUID.
RtlInitString	The RtlInitString routine initializes a counted string of 8-bit characters.
RtlInitStringEx	The RtlInitStringEx routine initializes a counted string of 8-bit characters.
RtlInitStringEx	The RtlInitStringEx routine initializes a counted string of 8-bit characters.
RtlInitUnicodeString	For more information, see the WdmlibRtlInitUnicodeStringEx function.
RtlInitUnicodeString	For more information, see the WdmlibRtlInitUnicodeStringEx function.
RtlInt64ToUnicodeString	The RtlInt64ToUnicodeString routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.
RtlInt64ToUnicodeString	The RtlInt64ToUnicodeString routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.
RtlInt8Add	Adds two values of type INT8.
RtlInt8Mult	Multiplies one value of type INT8 by another.
RtlInt8Sub	Subtracts one value of type INT8 from another.
RtlInt8ToUChar	Converts a value of type INT8 to a value of type UCHAR.
RtlInt8ToUInt	Converts a value of type INT8 to a value of type UINT.
RtlInt8ToUInt8	Converts a value of type INT8 to a value of type UINT8.
RtlInt8ToUIntPtr	Converts a value of type INT8 to a value of type UINT_PTR.
RtlInt8ToULong	Converts a value of type INT8 to a value of type ULONG.

TITLE	DESCRIPTION
RtlInt8ToULongLong	Converts a value of type INT8 to a value of type ULONGLONG.
RtlInt8ToULongPtr	Converts a value of type INT8 to a value of type ULONG_PTR.
RtlInt8ToUShort	Converts a value of type INT8 to a value of type USHORT.
RtlIntAdd	Adds two values of type INT.
RtlIntegerToUnicodeString	The RtlIntegerToUnicodeString routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.
RtlIntegerToUnicodeString	The RtlIntegerToUnicodeString routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.
RtlIntMult	Multiplies one value of type INT by another.
RtlIntPtrAdd	Adds two values of type INT_PTR.
RtlIntPtrMult	Multiplies one value of type INT_PTR by another.
RtlIntPtrSub	Subtracts one value of type INT_PTR from another.
RtlIntPtrToChar	Converts a value of type INT_PTR to a value of type CHAR.
RtlIntPtrToInt	Converts a value of type INT_PTR to a value of type INT.
RtlIntPtrToInt8	Converts a value of type INT_PTR to a value of type INT8.
RtlIntPtrToLong	Converts a value of type INT_PTR to a value of type LONG.
RtlIntPtrToLongPtr	Converts a value of type INT_PTR to a value of type LONG_PTR.
RtlIntPtrToShort	Converts a value of type INT_PTR to a value of type SHORT.
RtlIntPtrToUChar	Converts a value of type INT_PTR to a value of type UCHAR.
RtlIntPtrToUInt	Converts a value of type INT_PTR to a value of type UINT.
RtlIntPtrToUInt8	Converts a value of type INT_PTR to a value of type UINT8.
RtlIntPtrToUIntPtr	Converts a value of type INT_PTR to a value of type UINT_PTR.
RtlIntPtrToULong	Converts a value of type INT_PTR to a value of type ULONG.
RtlIntPtrToULongLong	Converts a value of type INT_PTR to a value of type ULONGLONG.

TITLE	DESCRIPTION
RtlIntPtrToULongPtr	Converts a value of type INT_PTR to a value of type ULONG_PTR.
RtlIntPtrToUnicodeString	The RtlIntPtrToUnicodeString routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.
RtlIntPtrToUnicodeString	The RtlIntPtrToUnicodeString routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.
RtlIntPtrToUShort	Converts a value of type INT_PTR to a value of type USHORT.
RtlIntSub	Subtracts one value of type INT from another.
RtlIntToChar	Converts a value of type INT to a value of type CHAR.
RtlIntToInt8	Converts a value of type INT to a value of type INT8.
RtlIntToShort	Converts a value of type INT to a value of type SHORT.
RtlIntToUChar	Converts a value of type INT to a value of type UCHAR.
RtlIntToUInt	Converts a value of type INT to a value of type UINT.
RtlIntToUInt8	Converts a value of type INT to a value of type UINT8.
RtlIntToULong	Converts a value of type INT to a value of type ULONG.
RtlIntToULongLong	Converts a value of type INT to a value of type ULONGLONG.
RtlIntToUShort	Converts a value of type INT to a value of type USHORT.
RtlIoDecodeMemIoResource	The RtlIoDecodeMemIoResource routine provides the address information that is contained in an IO_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.
RtlIoEncodeMemIoResource	The RtlIoEncodeMemIoResource routine updates an IO_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.
RtlIsNtDdiVersionAvailable	The RtlIsNtDdiVersionAvailable routine determines if a specified version of the Microsoft Windows device driver interface (DDI) is available.
RtlIsServicePackVersionInstalled	The RtlIsServicePackVersionInstalled routine determines if a specified service pack version of the Microsoft Windows device driver interface (DDI) is installed.
RtlIsStateSeparationEnabled	Checks if the SKU for the current context supports multiple sessions.

TITLE	DESCRIPTION
RtlLengthSecurityDescriptor	The RtlLengthSecurityDescriptor routine returns the size of a given security descriptor.
RtlLongAdd	Adds two values of type LONG.
RtlLongLongAdd	Adds two values of type LONGLONG.
RtlLongLongMult	Multiplies one value of type LONGLONG by another.
RtlLongLongSub	Subtracts one value of type LONGLONG from another.
RtlLongLongToChar	Converts a value of type LONGLONG to a value of type CHAR.
RtlLongLongToInt	Converts a value of type LONGLONG to a value of type INT.
RtlLongLongToInt8	Converts a value of type LONGLONG to a value of type INT8.
RtlLongLongToIntPtr	Converts a value of type LONGLONG to a value of type INT_PTR.
RtlLongLongToLong	Converts a value of type LONGLONG to a value of type LONG.
RtlLongLongToLongPtr	Converts a value of type LONGLONG to a value of type LONG_PTR.
RtlLongLongToShort	Converts a value of type LONGLONG to a value of type SHORT.
RtlLongLongToUChar	Converts a value of type LONGLONG to a value of type UCHAR.
RtlLongLongToUInt	Converts a value of type LONGLONG to a value of type UINT.
RtlLongLongToUInt8	Converts a value of type LONGLONG to a value of type UNIT8.
RtlLongLongToULong	Converts a value of type LONGLONG to a value of type ULONG.
RtlLongLongToULongLong	Converts a value of type LONGLONG to a value of type LONGLONG.
RtlLongLongToUShort	Converts a value of type LONGLONG to a value of type USHORT.
RtlLongMult	Multiplies one value of type LONG by another.
RtlLongPtrAdd	Adds two values of type LONG_PTR.
RtlLongPtrMult	Multiplies one value of type LONG_PTR by another.

TITLE	DESCRIPTION
RtlLongPtrSub	Subtracts one value of type LONG_PTR from another.
RtlLongPtrToChar	Converts a value of type LONG_PTR to a value of type CHAR.
RtlLongPtrToInt	Converts a value of type LONG_PTR to a value of type INT.
RtlLongPtrToInt8	Converts a value of type LONG_PTR to a value of type INT8.
RtlLongPtrToIntPtr	Converts a value of type LONG_PTR to a value of type INT_PTR.
RtlLongPtrToLong	Converts a value of type LONG_PTR to a value of type LONG.
RtlLongPtrToShort	Converts a value of type LONG_PTR to a value of type SHORT.
RtlLongPtrToUChar	Converts a value of type LONG_PTR to a value of type UCHAR.
RtlLongPtrToUInt	Converts a value of type LONG_PTR to a value of type UINT.
RtlLongPtrToUInt8	Converts a value of type LONG_PTR to a value of type UINT8.
RtlLongPtrToUIntPtr	Converts a value of type LONG_PTR to a value of type UINT_PTR.
RtlLongPtrToULong	Converts a value of type LONG_PTR to a value of type ULONG.
RtlLongPtrToULongLong	Converts a value of type LONG_PTR to a value of type ULONGLONG.
RtlLongPtrToULongPtr	Converts a value of type LONG_PTR to a value of type ULONG_PTR.
RtlLongPtrToUShort	Converts a value of type LONG_PTR to a value of type USHORT.
RtlLongSub	Subtracts one value of type LONG from another.
RtlLongToChar	Converts a value of type LONG to a value of type CHAR.
RtlLongToInt	Converts a value of type LONG to a value of type INT.
RtlLongToInt8	Converts a value of type LONG to a value of type INT8.
RtlLongToIntPtr	Converts a value of type LONG to a value of type INT_PTR.
RtlLongToShort	Converts a value of type LONG to a value of type SHORT.
RtlLongToUChar	Converts a value of type LONG to a value of type UCHAR.

TITLE	DESCRIPTION
RtlLongToUInt	Converts a value of type LONG to a value of type UINT.
RtlLongToUInt8	Converts a value of type LONG to a value of type UINT8.
RtlLongToUIntPtr	Converts a value of type LONG to a value of type UINT_PTR.
RtlLongToULong	Converts a value of type LONG to a value of type ULONG.
RtlLongToULongLong	Converts a value of type LONG to a value of type ULONGLONG.
RtlLongToULongPtr	Converts a value of type LONG to a value of type ULONG_PTR.
RtlLongToUShort	Converts a value of type LONG to a value of type USHORT.
RtlMapGenericMask	The RtlMapGenericMask routine determines the nongeneric access rights specified by an ACCESS_MASK.
RtlMoveMemory	The RtlMoveMemory routine copies the contents of a source memory block to a destination memory block, and supports overlapping source and destination memory blocks.
RtlNumberOfClearBits	The RtlNumberOfClearBits routine returns a count of the clear bits in a given bitmap variable.
RtlNumberOfSetBits	The RtlNumberOfSetBits routine returns a count of the set bits in a given bitmap variable.
RtlNumberOfSetBitsUlongPtr	The RtlNumberOfSetBitsUlongPtr routine returns the number of bits in the specified ULONG_PTR integer value that are set to one.
RtlPrefetchMemoryNonTemporal	The RtlPrefetchMemoryNonTemporal routine provides a hint to the processor that a buffer should be temporarily moved into the processor cache.
RtlPrefixUnicodeString	The RtlPrefixUnicodeString routine compares two Unicode strings to determine whether one string is a prefix of the other.
RtlPtrdiffTAdd	Adds two values of type PTRDIFF_T.
RtlPtrdiffTMult	Multiplies one value of type PTRDIFF_T by another.
RtlPtrdiffTSub	Subtracts one value of type PTRDIFF_T from another.
RtlQueryRegistryValues	The RtlQueryRegistryValues routine allows the caller to query several values from the registry subtree with a single call.
RtlQueryRegistryValueWithFallback	Retrieves a value entry for a registry key by using, a primary handle; if not found, uses the fallback handle.

TITLE	DESCRIPTION
RtlRunOnceBeginInitialize	The RtlRunOnceBeginInitialize routine begins a one-time initialization.
RtlRunOnceComplete	The RtlRunOnceComplete routine completes the one-time initialization began by RtlRunOnceBeginInitialize.
RtlRunOnceExecuteOnce	The RtlRunOnceExecuteOnce performs a one-time initialization.
RtlRunOnceInitialize	The RtlRunOnceInitialize routine initializes a RTL_RUN_ONCE structure.
RtlSecureZeroMemory	The RtlSecureZeroMemory routine fills a block of memory with zeros in a way that is guaranteed to be secure.
RtlSetAllBits	The RtlSetAllBits routine sets all bits in a given bitmap variable.
RtlSetBit	The RtlSetBit routine sets the specified bit in a bitmap to one.
RtlSetBits	The RtlSetBits routine sets all bits in a given range of a given bitmap variable.
RtlSetDaclSecurityDescriptor	The RtlSetDaclSecurityDescriptor routine sets the DACL information of an absolute-format security descriptor, or if there is already a DACL present in the security descriptor, it is superseded.
RtlShortAdd	Adds two values of type SHORT.
RtlShortMult	Multiplies one value of type SHORT by another.
RtlShortSub	Subtracts one value of type SHORT from another.
RtlShortToChar	Converts a value of type SHORT to a value of type CHAR.
RtlShortToDWordPtr	Converts a value of type SHORT to a value of type DWORD_PTR.
RtlShortToInt8	Converts a value of type SHORT to a value of type INT8.
RtlShortToUChar	Converts a value of type SHORT to a value of type UCHAR.
RtlShortToUInt	Converts a value of type SHORT to a value of type UINT.
RtlShortToUInt8	Converts a value of type SHORT to a value of type UINT8.
RtlShortToUIntPtr	Converts a value of type SHORT to a value of type UINT_PTR.
RtlShortToULong	Converts a value of type SHORT to a value of type ULONG.
RtlShortToULongLong	Converts a value of type SHORT to a value of type ULONGLONG.

TITLE	DESCRIPTION
RtlShortToULongPtr	Converts a value of type SHORT to a value of type ULONG_PTR.
RtlShortToUShort	Converts a value of type SHORT to a value of type USHORT.
RtlSizeTAdd	Adds two values of type SIZE_T.
RtlSizeTMult	Multiplies one value of type SIZE_T by another.
RtlSizeTSub	Subtracts one value of type SIZE_T from another.
RtlSSIZETAdd	Adds two values of type SSIZE_T.
RtlSSIZETMult	Multiplies one value of type SSIZE_T by another.
RtlSSIZETSub	Subtracts one value of type SSIZE_T from another.
RtlStringCbCatA	The RtlStringCbCatW and RtlStringCbCatA functions concatenate two byte-counted strings.
RtlStringCbCatExA	The RtlStringCbCatExW and RtlStringCbCatExA functions concatenate two byte-counted strings.
RtlStringCbCatExW	The RtlStringCbCatExW and RtlStringCbCatExA functions concatenate two byte-counted strings.
RtlStringCbCatNA	The RtlStringCbCatNW and RtlStringCbCatNA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNExA	The RtlStringCbCatNExW and RtlStringCbCatNExA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNExW	The RtlStringCbCatNExW and RtlStringCbCatNExA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNW	The RtlStringCbCatNW and RtlStringCbCatNA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatW	The RtlStringCbCatW and RtlStringCbCatA functions concatenate two byte-counted strings.
RtlStringCbCopyA	The RtlStringCbCopyW and RtlStringCbCopyA functions copy a byte-counted string into a buffer.
RtlStringCbCopyExA	The RtlStringCbCopyExW and RtlStringCbCopyExA functions copy a byte-counted string into a buffer.
RtlStringCbCopyExW	The RtlStringCbCopyExW and RtlStringCbCopyExA functions copy a byte-counted string into a buffer.

TITLE	DESCRIPTION
RtlStringCbCopyNA	The RtlStringCbCopyNW and RtlStringCbCopyNA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNExA	The RtlStringCbCopyNExW and RtlStringCbCopyNExA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNExW	The RtlStringCbCopyNExW and RtlStringCbCopyNExA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNW	The RtlStringCbCopyNW and RtlStringCbCopyNA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyUnicodeString	The RtlStringCbCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCbCopyUnicodeStringEx	The RtlStringCbCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCbCopyW	The RtlStringCbCopyW and RtlStringCbCopyA functions copy a byte-counted string into a buffer.
RtlStringCbLengthA	The RtlStringCbLengthW and RtlStringCbLengthA functions determine the length, in bytes, of a supplied string.
RtlStringCbLengthW	The RtlStringCbLengthW and RtlStringCbLengthA functions determine the length, in bytes, of a supplied string.
RtlStringCbPrintfA	The RtlStringCbPrintfW and RtlStringCbPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfExA	The RtlStringCbPrintfExW and RtlStringCbPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfExW	The RtlStringCbPrintfExW and RtlStringCbPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfW	The RtlStringCbPrintfW and RtlStringCbPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfA	The RtlStringCbVPrintfW and RtlStringCbVPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.

TITLE	DESCRIPTION
RtlStringCbVPrintfExA	The RtlStringCbVPrintfExW and RtlStringCbVPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfExW	The RtlStringCbVPrintfExW and RtlStringCbVPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfW	The RtlStringCbVPrintfW and RtlStringCbVPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchCatA	The RtlStringCchCatW and RtlStringCchCatA functions concatenate two character-counted strings.
RtlStringCchCatExA	The RtlStringCchCatExW and RtlStringCchCatExA functions concatenate two character-counted strings.
RtlStringCchCatExW	The RtlStringCchCatExW and RtlStringCchCatExA functions concatenate two character-counted strings.
RtlStringCchCatNA	The RtlStringCchCatNW and RtlStringCchCatNA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNExA	The RtlStringCchCatNExW and RtlStringCchCatNExA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNExW	The RtlStringCchCatNExW and RtlStringCchCatNExA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNW	The RtlStringCchCatNW and RtlStringCchCatNA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatW	The RtlStringCchCatW and RtlStringCchCatA functions concatenate two character-counted strings.
RtlStringCchCopyA	The RtlStringCchCopyW and RtlStringCchCopyA functions copy a null-terminated source string into a destination buffer of specified length.
RtlStringCchCopyExA	The RtlStringCchCopyExW and RtlStringCchCopyExA functions copy a character-counted string into a buffer.
RtlStringCchCopyExW	The RtlStringCchCopyExW and RtlStringCchCopyExA functions copy a character-counted string into a buffer.
RtlStringCchCopyNA	The RtlStringCchCopyNW and RtlStringCchCopyNA functions copy a character-counted string to a buffer while limiting the size of the copied string.

TITLE	DESCRIPTION
RtlStringCchCopyNExA	The RtlStringCchCopyNExW and RtlStringCchCopyNExA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNExW	The RtlStringCchCopyNExW and RtlStringCchCopyNExA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNW	The RtlStringCchCopyNW and RtlStringCchCopyNA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyUnicodeString	The RtlStringCchCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyUnicodeStringEx	The RtlStringCchCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyW	The RtlStringCchCopyW and RtlStringCchCopyA functions copy a null-terminated source string into a destination buffer of specified length.
RtlStringCchLengthA	The RtlStringCchLengthW and RtlStringCchLengthA functions determine the length, in characters, of a supplied string.
RtlStringCchLengthW	The RtlStringCchLengthW and RtlStringCchLengthA functions determine the length, in characters, of a supplied string.
RtlStringCchPrintfA	The RtlStringCchPrintfW and RtlStringCchPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfExA	The RtlStringCchPrintfExW and RtlStringCchPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfExW	The RtlStringCchPrintfExW and RtlStringCchPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfW	The RtlStringCchPrintfW and RtlStringCchPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfA	The RtlStringCchVPrintfW and RtlStringCchVPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfExA	The RtlStringCchVPrintfExW and RtlStringCchVPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.

TITLE	DESCRIPTION
RtlStringCchVPrintfExW	The RtlStringCchVPrintfExW and RtlStringCchVPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfW	The RtlStringCchVPrintfW and RtlStringCchVPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringFromGUID	The RtlStringFromGUID routine converts a given GUID from binary format into a Unicode string.
RtlTestBit	The RtlTestBit routine returns the value of a bit in a bitmap.
RtlTimeFieldsToTime	The RtlTimeFieldsToTime routine converts TIME_FIELDS information to a system time value.
RtlTimeToTimeFields	The RtlTimeToTimeFields routine converts system time into a TIME_FIELDS structure.
RtlUInt8Add	Adds two values of type UINT8.
RtlUInt8Mult	Multiplies one value of type UINT8 by another.
RtlUInt8Sub	The RtlUInt8Sub routine subtracts one value of type UINT8 from another.
RtlUInt8ToChar	Converts a value of type UINT8 to a value of type CHAR.
RtlUInt8ToInt8	Converts a value of type UINT8 to a value of type INT8.
RtlUIntAdd	Adds two values of type UINT.
RtlUIntMult	Multiplies one value of type UINT by another.
RtlUIntPtrAdd	Adds two values of type UINT_PTR.
RtlUIntPtrMult	Multiplies one value of type UINT_PTR by another.
RtlUIntPtrSub	Subtracts one value of type UINT_PTR from another.
RtlUIntPtrToChar	Converts a value of type UINT_PTR to a value of type CHAR.
RtlUIntPtrToInt	Converts a value of type UINT_PTR to a value of type INT.
RtlUIntPtrToInt16	Converts a value of type UINT_PTR to a value of type INT16.
RtlUIntPtrToInt8	Converts a value of type UINT_PTR to a value of type INT8.
RtlUIntPtrToIntPtr	Converts a value of type UINT_PTR to a value of type INT_PTR.
RtlUIntPtrToLong	Converts a value of type UINT_PTR to a value of type LONG.

TITLE	DESCRIPTION
RtlUIntPtrToLongLong	Converts a value of type UINT_PTR to a value of type LONGLONG.
RtlUIntPtrToLongPtr	Converts a value of type UINT_PTR to a value of type LONG_PTR.
RtlUIntPtrToShort	Converts a value of type UINT_PTR to a value of type SHORT.
RtlUIntPtrToUChar	Converts a value of type UINT_PTR to a value of type UCHAR.
RtlUIntPtrToUInt	Converts a value of type UINT_PTR to a value of type UINT.
RtlUIntPtrToUInt16	Converts a value of type UINT_PTR to a value of type UINT16.
RtlUIntPtrToUInt8	Converts a value of type UINT_PTR to a value of type UINT8.
RtlUIntPtrToULong	Converts a value of type UINT_PTR to a value of type LONG.
RtlUIntPtrToUShort	Converts a value of type UINT_PTR to a value of type USHORT.
RtlUIntSub	Subtracts one value of type UINT from another.
RtlUIntToChar	Converts a value of type UINT to a value of type CHAR.
RtlUIntToInt	Converts a value of type UINT to a value of type INT.
RtlUIntToInt8	Converts a value of type UINT to a value of type INT8.
RtlUIntToIntPtr	Converts a value of type UINT to a value of type INT_PTR.
RtlUIntToLong	Converts a value of type UINT to a value of type LONG.
RtlUIntToLongPtr	Converts a value of type UINT to a value of type LONG_PTR.
RtlUIntToShort	Converts a value of type UINT to a value of type SHORT.
RtlUIntToUChar	Converts a value of type UINT to a value of type UCHAR.
RtlUIntToUInt8	Converts a value of type UINT to a value of type UINT8.
RtlUIntToUShort	Converts a value of type UINT to a value of type USHORT.
RtlULongAdd	Adds two values of type ULONG.
RtlUlongByteSwap	The RtlUlongByteSwap routine reverses the ordering of the four bytes in a 32-bit unsigned integer value.
RtlULongLongAdd	Adds two values of type ULONGLONG.

TITLE	DESCRIPTION
RtlUlonglongByteSwap	The RtlUlonglongByteSwap routine reverses the ordering of the eight bytes in a 64-bit unsigned integer value.
RtlULongLongMult	Multiplies one value of type ULONGLONG by another.
RtlULongLongSub	Subtracts one value of type ULONGLONG from another.
RtlULongLongToChar	Converts a value of type ULONGLONG to a value of type CHAR.
RtlULongLongToInt	Converts a value of type ULONGLONG to a value of type INT.
RtlULongLongToInt8	Converts a value of type ULONGLONG to a value of type INT8.
RtlULongLongToLong	Converts a value of type ULONGLONG to a value of type LONG.
RtlULongLongToLongLong	Converts a value of type ULONGLONG to a value of type LONGLONG.
RtlULongLongToLongPtr	Converts a value of type ULONGLONG to a value of type LONG_PTR.
RtlULongLongToShort	Converts a value of type ULONGLONG to a value of type SHORT.
RtlULongLongToUChar	Converts a value of type ULONGLONG to a value of type UCHAR.
RtlULongLongToUInt	Converts a value of type ULONGLONG to a value of type UINT.
RtlULongLongToUInt8	Converts a value of type ULONGLONG to a value of type UINT8.
RtlULongLongToUIntPtr	Converts a value of type ULONGLONG to a value of type UINT_PTR.
RtlULongLongToULong	Converts a value of type ULONGLONG to a value of type ULONG.
RtlULongLongToULongPtr	Converts a value of type ULONGLONG to a value of type ULONG_PTR.
RtlULongLongToUShort	Converts a value of type ULONGLONG to a value of type USHORT.
RtlULongMult	Multiplies one value of type ULONG by another.
RtlULongPtrAdd	Adds two values of type ULONG_PTR.
RtlULongPtrMult	Multiplies one value of type ULONG_PTR by another.

TITLE	DESCRIPTION
RtlULongPtrSub	Subtracts one value of type ULONG_PTR from another.
RtlULongPtrToChar	Converts a value of type ULONG_PTR to a value of type CHAR.
RtlULongPtrToInt	Converts a value of type ULONG_PTR to a value of type INT.
RtlULongPtrToInt8	Converts a value of type ULONG_PTR to a value of type INT8.
RtlULongPtrToIntPtr	Converts a value of type ULONG_PTR to a value of type INT_PTR.
RtlULongPtrToLong	Converts a value of type ULONG_PTR to a value of type LONG.
RtlULongPtrToLongLong	Converts a value of type ULONG_PTR to a value of type LONGLONG.
RtlULongPtrToLongPtr	Converts a value of type ULONG_PTR to a value of type LONG_PTR.
RtlULongPtrToShort	Converts a value of type ULONG_PTR to a value of type SHORT.
RtlULongPtrToUChar	Converts a value of type ULONG_PTR to a value of type UCHAR.
RtlULongPtrToUInt	Converts a value of type ULONG_PTR to a value of type UINT.
RtlULongPtrToUInt8	Converts a value of type ULONG_PTR to a value of type UINT8.
RtlULongPtrToUIntPtr	Converts a value of type ULONG_PTR to a value of type UINT_PTR.
RtlULongPtrToULong	Converts a value of type ULONG_PTR to a value of type ULONG.
RtlULongPtrToUShort	Converts a value of type ULONG_PTR to a value of type USHORT.
RtlULongSub	Subtracts one value of type ULONG from another.
RtlULongToChar	Converts a value of type ULONG to a value of type CHAR.
RtlULongToInt	Converts a value of type ULONG to a value of type INT.
RtlULongToInt8	Converts a value of type ULONG to a value of type INT8.
RtlULongToIntPtr	Converts a value of type ULONG to a value of type INT_PTR.
RtlULongToLong	Converts a value of type ULONG to a value of type LONG.

TITLE	DESCRIPTION
RtlULongToLongPtr	Converts a value of type ULONG to a value of type LONG_PTR.
RtlULongToShort	Converts a value of type ULONG to a value of type SHORT.
RtlULongToUChar	Converts a value of type ULONG to a value of type UCHAR.
RtlULongToUInt	Converts a value of type ULONG to a value of type UINT.
RtlULongToUInt8	Converts a value of type ULONG_PTR to a value of type UINT8.
RtlULongToUIntPtr	Converts a value of type ULONG_PTR to a value of type UINT_PTR.
RtlULongToUShort	Converts a value of type ULONG to a value of type USHORT.
RtlUnalignedStringCbLengthW	The RtlUnalignedStringCbLengthW function is a version of the RtlStringCbLength function that accepts an unaligned pointer to a string of Unicode characters.
RtlUnalignedStringCchLengthW	The RtlUnalignedStringCchLengthW function is a version of the RtlStringCchLength function that accepts an unaligned pointer to a string of Unicode characters.
RtlUnicodeStringCat	The RtlUnicodeStringCat function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatEx	The RtlUnicodeStringCatEx function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatString	The RtlUnicodeStringCatString function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCatStringEx	The RtlUnicodeStringCatStringEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCbCatN	The RtlUnicodeStringCbCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCbCatNEx	The RtlUnicodeStringCbCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCbCatStringN	The RtlUnicodeStringCbCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.

TITLE	DESCRIPTION
RtlUnicodeStringCbCatStringNEx	The RtlUnicodeStringCbCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCbCopyN	The RtlUnicodeStringCbCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCbCopyNEx	The RtlUnicodeStringCbCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringN	The RtlUnicodeStringCbCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringNEx	The RtlUnicodeStringCbCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCchCatN	The RtlUnicodeStringCchCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatNEx	The RtlUnicodeStringCchCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatStringN	The RtlUnicodeStringCchCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCatStringNEx	The RtlUnicodeStringCchCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCopyN	The RtlUnicodeStringCchCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyNEx	The RtlUnicodeStringCchCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyStringN	The RtlUnicodeStringCchCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCchCopyStringNEx	The RtlUnicodeStringCchCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

TITLE	DESCRIPTION
RtlUnicodeStringCopy	The RtlUnicodeStringCopy function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyEx	The RtlUnicodeStringCopyEx function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyString	The RtlUnicodeStringCopyString function copies a string into a UNICODE_STRING structure.
RtlUnicodeStringCopyStringEx	The RtlUnicodeStringCopyStringEx function copies a string into a UNICODE_STRING structure.
RtlUnicodeStringInit	The RtlUnicodeStringInit function initializes a UNICODE_STRING structure.
RtlUnicodeStringInitEx	The RtlUnicodeStringInitEx function initializes a UNICODE_STRING structure.
RtlUnicodeStringPrintf	The RtlUnicodeStringPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringPrintfEx	The RtlUnicodeStringPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringToAnsiSize	The RtlUnicodeStringToAnsiSize routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.
RtlUnicodeStringToAnsiString	The RtlUnicodeStringToAnsiString routine converts a given Unicode string into an ANSI string.
RtlUnicodeStringToInteger	The RtlUnicodeStringToInteger routine converts a Unicode string representation of a number to the equivalent integer value.
RtlUnicodeStringToInteger	The RtlUnicodeStringToInteger routine converts a Unicode string representation of a number to the equivalent integer value.
RtlUnicodeStringValidate	The RtlUnicodeStringValidate function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringValidateEx	The RtlUnicodeStringValidateEx function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringVPrintf	The RtlUnicodeStringVPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

TITLE	DESCRIPTION
RtlUnicodeStringVPrintfEx	The RtlUnicodeStringVPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeToUTF8N	The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.
RtlUnicodeToUTF8N	The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.
RtlUpcaseUnicodeChar	The RtlUpcaseUnicodeChar routine converts the specified Unicode character to uppercase.
RtlUpcaseUnicodeString	The RtlUpcaseUnicodeString routine converts a copy of the source string to uppercase and writes the converted string in the destination buffer.
RtlUpperChar	The RtlUpperChar routine converts the specified character to uppercase.
RtlUpperString	The RtlUpperString routine copies the given SourceString to the DestinationString buffer, converting it to uppercase.
RtlUShortAdd	Adds two values of type USHORT.
RtlUshortByteSwap	The RtlUshortByteSwap routine reverses the ordering of the two bytes in a 16-bit unsigned integer value.
RtlUShortMult	Multiplies one value of type USHORT by another.
RtlUShortSub	Subtracts one value of type USHORT from another.
RtlUShortToChar	Converts a value of type USHORT to a value of type CHAR.
RtlUShortToInt8	Converts a value of type USHORT to a value of type INT8.
RtlUShortToShort	Converts a value of type USHORT to a value of type SHORT.
RtlUShortToUChar	Converts a value of type USHORT to a value of type UCHAR.
RtlUShortToUInt8	Converts a value of type USHORT to a value of type UINT8.
RtlUTF8ToUnicodeN	The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.
RtlUTF8ToUnicodeN	The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.
RtlValidateCorrelationVector	Validates the specified correlation vector to check whether it conforms to the Correlation Vector Specification (v2).

TITLE	DESCRIPTION
RtlValidRelativeSecurityDescriptor	The RtlValidRelativeSecurityDescriptor routine checks the validity of a self-relative security descriptor.
RtlValidSecurityDescriptor	The RtlValidSecurityDescriptor routine checks a given security descriptor's validity.
RtlVerifyVersionInfo	The RtlVerifyVersionInfo routine compares a specified set of operating system version requirements to the corresponding attributes of the currently running version of the operating system.
RtlVolumeDeviceToDosName	The RtlVolumeDeviceToDosName routine is obsolete for Windows XP and later versions of Windows. Use IoVolumeDeviceToDosName instead.RtlVolumeDeviceToDosName returns the MS-DOS path for a specified device object that represents a file system volume.
RtlWriteRegistryValue	The RtlWriteRegistryValue routine writes caller-supplied data into the registry along the specified relative path at the given value name.
RtlxAnsiStringToUnicodeSize	The RtlxAnsiStringToUnicodeSize routine returns the number of bytes that are required for a null-terminated Unicode string that is equivalent to a specified ANSI string.
RtlxUnicodeStringToAnsiSize	The RtlxUnicodeStringToAnsiSize routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.
RtlZeroMemory	The RtlZeroMemory routine fills a block of memory with zeros, given a pointer to the block and the length, in bytes, to be filled.
SeAccessCheck	The SeAccessCheck routine determines whether the requested access rights can be granted to an object protected by a security descriptor and an object owner.
SeAssignSecurity	The SeAssignSecurity routine builds a self-relative security descriptor for a new object, given the security descriptor of its parent directory and any originally requested security for the object.
SeAssignSecurityEx	The SeAssignSecurityEx routine builds a self-relative security descriptor for a new object given the following optional parameters:_a security descriptor of the object's parent directory, an explicit security descriptor for the object, and the object type.
SeDeassignSecurity	The SeDeassignSecurity routine deallocates the memory associated with a security descriptor that was assigned using SeAssignSecurity.
SeFreePrivileges	The SeFreePrivileges routine frees a privilege set returned by SeAccessCheck.

TITLE	DESCRIPTION
SeSinglePrivilegeCheck	The SeSinglePrivilegeCheck routine checks for the passed privilege value in the context of the current thread.
SET_D3COLD_SUPPORT	The SetD3ColdSupport routine enables or disables transitions to the D3cold device power state.
SeValidSecurityDescriptor	The SeValidSecurityDescriptor routine returns whether a given security descriptor is structurally valid.
SILO_CONTEXT_CLEANUP_CALLBACK	This callback is invoked when the context object reaches a reference count of zero.
SILO_MONITOR_CREATE_CALLBACK	This is callback is invoked when a new silo is created.
SILO_MONITOR_TERMINATE_CALLBACK	This callback is invoked when a silo is terminated.
TmCommitComplete	The TmCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction.
TmCommitEnlistment	The TmCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
TmCommitTransaction	The TmCommitTransaction routine initiates a commit operation for a specified transaction.
TmCreateEnlistment	The TmCreateEnlistment routine creates a new enlistment object for a transaction.
TmDereferenceEnlistmentKey	The TmDereferenceEnlistmentKey routine decrements the reference count for the key of a specified enlistment object.
TmEnableCallbacks	The TmEnableCallbacks routine enables a callback routine that receives transaction notifications.
TmGetTransactionId	The TmGetTransactionId routine retrieves a transaction object's unit of work (UOW) identifier.
TmInitializeTransactionManager	The TmInitializeTransactionManager routine initializes a transaction manager object.
TmIsTransactionActive	The TmIsTransactionActive routine indicates whether a specified transaction is in its active state.
TmPrepareComplete	The TmPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
TmPrepareEnlistment	The TmPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
TmPrePrepareComplete	The TmPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

TITLE	DESCRIPTION
TmPrePrepareEnlistment	The TmPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
TmReadOnlyEnlistment	The TmReadOnlyEnlistment routine sets a specified enlistment to be read-only.
TmRecoverEnlistment	The TmRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.
TmRecoverResourceManager	The TmRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
TmRecoverTransactionManager	The TmRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
TmReferenceEnlistmentKey	The TmReferenceEnlistmentKey routine increments the reference count for the key of a specified enlistment object and retrieves the key.
TmRenameTransactionManager	The TmRenameTransactionManager routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.
TmRequestOutcomeEnlistment	The TmRequestOutcomeEnlistment routine asks KTM to try to provide an immediate outcome (commit or rollback) for the transaction that is associated with a specified enlistment.
TmRollbackComplete	The TmRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
TmRollbackEnlistment	The TmRollbackEnlistment routine rolls back a specified enlistment.
TmRollbackTransaction	The TmRollbackTransaction routine initiates a rollback operation for a specified transaction.
TmSinglePhaseReject	The TmSinglePhaseReject routine informs KTM that the calling resource manager will not support a single-phase commit operation for a specified enlistment.
TRANSLATE_BUS_ADDRESS	The TranslateBusAddress routine translates addresses on the parent bus to logical addresses.
VslCreateSecureSection	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
VslDeleteSecureSection	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.

TITLE	DESCRIPTION
WdmlibIoConnectInterruptEx	The WdmlibIoConnectInterruptEx function registers an interrupt-handling routine for a device's interrupts.
WdmlibIoCreateDeviceSecure	The WdmlibIoCreateDeviceSecure function (or IoCreateDeviceSecure) creates a named device object and applies the specified security settings.
WdmlibIoDisconnectInterruptEx	The WdmlibIoDisconnectInterruptEx function unregisters an interrupt service routine (ISR) that was registered by a previous call to the WdmlibIoConnectInterruptEx function.
WdmlibIoGetAffinityInterrupt	The WdmlibIoGetAffinityInterrupt function gets the group affinity of an interrupt object.
WdmlibIoValidateDeviceIoControlAccess	The WdmlibIoValidateDeviceIoControlAccess function verifies that the sender of an IRP_MJ_DEVICE_CONTROL or IRP_MJ_FILE_SYSTEM_CONTROL IRP has the specified access to the device object.
WdmlibProcgrpInitialize	The WdmlibProcgrpInitialize function initializes the Processor Group (ProcGrp) compatibility library.
WdmlibRtlInitUnicodeStringEx	The WdmlibRtlInitUnicodeStringEx function initializes a counted string of Unicode characters.
WMI_EXECUTE_METHOD_CALLBACK	The DpWmiExecuteMethod routine executes a method associated with a data block. This routine is optional.
WMI_FUNCTION_CONTROL_CALLBACK	The DpWmiFunctionControl routine enables or disables notification of events, and enables or disables data collection for data blocks that the driver registered as expensive to collect. This routine is optional.
WMI_QUERY_DATABLOCK_CALLBACK	The DpWmiQueryDataBlock routine returns either a single instance or all instances of a data block. This routine is required.
WMI_QUERY_REGINFO_CALLBACK	The DpWmiQueryReginfo routine provides information about the data blocks and event blocks to be registered by a driver. This routine is required.
WMI_SET_DATABLOCK_CALLBACK	The DpWmiSetDataBlock routine changes all data items in a single instance of a data block. This routine is optional.
WMI_SET_DATAITEM_CALLBACK	The DpWmiSetDataItem routine changes a single data item in an instance of a data block. This routine is optional.
WmiCompleteRequest	The WmiCompleteRequest routine indicates that a driver has finished processing a WMI request in a DpWmiXxx routine.
WmiFireEvent	The WmiFireEvent routine sends an event to WMI for delivery to data consumers that have requested notification of the event.

TITLE	DESCRIPTION
WmiQueryTraceInformation	The WmiQueryTraceInformation routine returns information about a WMI event trace.
WmiSystemControl	The WmiSystemControl routine is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.
WmiTraceMessage	The WmiTraceMessage routine adds a message to the output log of a WPP software tracing session.
WmiTraceMessageVa	The WmiTraceMessageVa routine adds a message to the output log of a WPP software tracing session.
WRITE_PORT_BUFFER_UCHAR	The WRITE_PORT_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified port.
WRITE_PORT_BUFFER_UCHAR	The WRITE_PORT_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified port.
WRITE_PORT_BUFFER_ULONG	The WRITE_PORT_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified port address.
WRITE_PORT_BUFFER_ULONG	The WRITE_PORT_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified port address.
WRITE_PORT_BUFFER_USHORT	The WRITE_PORT_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified port address.
WRITE_PORT_BUFFER_USHORT	The WRITE_PORT_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified port address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.

TITLE	DESCRIPTION
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG64	The WRITE_REGISTER_BUFFER_ULONG64 function writes a number of ULONG64 values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.

TITLE	DESCRIPTION
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.
ZwAllocateLocallyUniqueId	The ZwAllocateLocallyUniqueId routine allocates a locally unique identifier (LUID).
ZwClose	The ZwClose routine closes an object handle.
ZwCommitComplete	The ZwCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction's data.
ZwCommitEnlistment	The ZwCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
ZwCommitTransaction	The ZwCommitTransaction routine initiates a commit operation for a specified transaction.
ZwCreateDirectoryObject	The ZwCreateDirectoryObject routine creates or opens an object-directory object.
ZwCreateEnlistment	The ZwCreateEnlistment routine creates a new enlistment object for a transaction.
ZwCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
ZwCreateKey	The ZwCreateKey routine creates a new registry key or opens an existing one.
ZwCreateKeyTransacted	The ZwCreateKeyTransacted routine creates a new registry key or opens an existing one, and it associates the key with a transaction.
ZwCreateResourceManager	The ZwCreateResourceManager routine creates a resource manager object.

TITLE	DESCRIPTION
ZwCreateSection	The ZwCreateSection routine creates a section object.
ZwCreateTransaction	The ZwCreateTransaction routine creates a transaction object.
ZwCreateTransactionManager	The ZwCreateTransactionManager routine creates a new transaction manager object.
ZwDeleteKey	The ZwDeleteKey routine deletes an open key from the registry.
ZwDeleteValueKey	The ZwDeleteValueKey routine deletes a value entry matching a name from an open key in the registry. If no such entry exists, an error is returned.
ZwDeviceIoControlFile	The ZwDeviceIoControlFile routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.
ZwEnumerateKey	The ZwEnumerateKey routine returns information about a subkey of an open registry key.
ZwEnumerateTransactionObject	The ZwEnumerateTransactionObject routine enumerates the KTM objects on a computer.
ZwEnumerateValueKey	The ZwEnumerateValueKey routine gets information about the value entries of an open key.
ZwFlushKey	The ZwFlushKey routine forces a registry key to be committed to disk.
ZwGetNotificationResourceManager	The ZwGetNotificationResourceManager routine retrieves the next transaction notification from a specified resource manager's notification queue.
ZwLoadDriver	The ZwLoadDriver routine loads a driver into the system.
ZwMakeTemporaryObject	The ZwMakeTemporaryObject routine changes the attributes of an object to make it temporary.
ZwMapViewOfSection	The ZwMapViewOfSection routine maps a view of a section into the virtual address space of a subject process.
ZwOpenEnlistment	The ZwOpenEnlistment routine obtains a handle to an existing enlistment object.
ZwOpenEvent	The ZwOpenEvent routine opens a handle to an existing named event object with the specified desired access.
ZwOpenFile	The ZwOpenFile routine opens an existing file, directory, device, or volume.
ZwOpenKey	The ZwOpenKey routine opens an existing registry key.

TITLE	DESCRIPTION
ZwOpenKeyEx	The ZwOpenKeyEx routine opens an existing registry key.
ZwOpenKeyTransacted	The ZwOpenKeyTransacted routine opens an existing registry key and associates the key with a transaction.
ZwOpenKeyTransactedEx	The ZwOpenKeyTransactedEx routine opens an existing registry key and associates the key with a transaction.
ZwOpenProcess	The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.
ZwOpenResourceManager	The ZwOpenResourceManager routine returns a handle to an existing resource manager object.
ZwOpenSection	The ZwOpenSection routine opens a handle for an existing section object.
ZwOpenSymbolicLinkObject	The ZwOpenSymbolicLinkObject routine opens an existing symbolic link.
ZwOpenTransaction	The ZwOpenTransaction routine obtains a handle to an existing transaction object.
ZwOpenTransactionManager	The ZwOpenTransactionManager routine obtains a handle to an existing transaction manager object.
ZwPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.
ZwPrepareComplete	The ZwPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
ZwPrepareEnlistment	The ZwPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
ZwPrePrepareComplete	The ZwPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.
ZwPrePrepareEnlistment	The ZwPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
ZwQueryFullAttributesFile	The ZwQueryFullAttributesFile routine supplies network open information for the specified file.
ZwQueryInformationEnlistment	The ZwQueryInformationEnlistment routine retrieves information about a specified enlistment object.
ZwQueryInformationFile	The ZwQueryInformationFile routine returns various kinds of information about a file object.
ZwQueryInformationResourceManager	The ZwQueryInformationResourceManager routine retrieves information about a specified resource manager object.

TITLE	DESCRIPTION
ZwQueryInformationTransaction	The ZwQueryInformationTransaction routine retrieves information about a specified transaction.
ZwQueryInformationTransactionManager	The ZwQueryInformationTransactionManager routine retrieves information about a specified transaction manager object.
ZwQueryKey	The ZwQueryKey routine provides information about the class of a registry key, and the number and sizes of its subkeys.
ZwQuerySymbolicLinkObject	The ZwQuerySymbolicLinkObject routine returns a Unicode string that contains the target of a symbolic link.
ZwQueryValueKey	The ZwQueryValueKey routine returns a value entry for a registry key.
ZwQueryVolumeInformationFile	The ZwQueryVolumeInformationFile routine retrieves information about the volume associated with a given file, directory, storage device, or volume.
ZwReadFile	The ZwReadFile routine reads data from an open file.
ZwReadOnlyEnlistment	The ZwReadOnlyEnlistment routine sets a specified enlistment to be read-only.
ZwRecoverEnlistment	The ZwRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.
ZwRecoverResourceManager	The ZwRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
ZwRecoverTransactionManager	The ZwRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
ZwRollbackComplete	The ZwRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
ZwRollbackEnlistment	The ZwRollbackEnlistment routine rolls back the transaction that is associated with a specified enlistment.
ZwRollbackTransaction	The ZwRollbackTransaction routine initiates a rollback operation for a specified transaction.
ZwRollforwardTransactionManager	The ZwRollforwardTransactionManager routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.
ZwSetInformationEnlistment	The ZwSetInformationEnlistment routine sets information for a specified enlistment object.

TITLE	DESCRIPTION
ZwSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
ZwSetInformationResourceManager	The ZwSetInformationResourceManager routine is not used.
ZwSetInformationThread	The ZwSetInformationThread routine sets the priority of a thread.
ZwSetInformationTransaction	The ZwSetInformationTransaction routine sets information for a specified transaction.
ZwSetValueKey	The ZwSetValueKey routine creates or replaces a registry key's value entry.
ZwSinglePhaseReject	The ZwSinglePhaseReject routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.
ZwTerminateProcess	The ZwTerminateProcess routine terminates a process and all of its threads.
ZwUnloadDriver	The ZwUnloadDriver routine unloads a driver from the system.
ZwUnmapViewOfSection	The ZwUnmapViewOfSection routine unmaps a view of a section from the virtual address space of a subject process.
ZwWriteFile	The ZwWriteFile routine writes data to an open file.

Structures

TITLE	DESCRIPTION
DMA_ADAPTER	The DMA_ADAPTER structure describes a system-defined interface to a DMA controller for a given device. A driver calls IoGetDmaAdapter to obtain this structure.
DMA_OPERATIONS	The DMA_OPERATIONS structure provides a table of pointers to functions that control the operation of a DMA controller.
IO_STATUS_BLOCK	A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.
AUX_MODULE_BASIC_INFO	The AUX_MODULE_BASIC_INFO structure contains basic information about a loaded image module.
AUX_MODULE_EXTENDED_INFO	The AUX_MODULE_EXTENDED_INFO structure contains extended information about a loaded image module.
BDCB_IMAGE_INFORMATION	The BDCB_IMAGE_INFORMATION structure describes information about a boot-start driver that is about to be initialized, provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.

TITLE	DESCRIPTION
BDCB_STATUS_UPDATE_CONTEXT	The BDCB_STATUS_UPDATE_CONTEXT structure describes a status update provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.
BOOTDISK_INFORMATION	The BOOTDISK_INFORMATION structure contains basic information describing the boot and system disks.
BOOTDISK_INFORMATION_EX	The BOOTDISK_INFORMATION_EX structure contains extended information describing the boot and system disks.
BUS_INTERFACE_STANDARD	The BUS_INTERFACE_STANDARD interface structure enables device drivers to make direct calls to parent bus driver routines. This structure defines the GUID_BUS_INTERFACE_STANDARD interface.
CLFS_LOG_NAME_INFORMATION	The CLFS_LOG_NAME_INFORMATION structure holds the name of a Common Log File System (CLFS) stream or log.
CLFS_MGMT_CLIENT_REGISTRATION	The CLFS_MGMT_CLIENT_REGISTRATION structure is given to CLFS management by clients who manage their own logs.
CLFS_MGMT_POLICY	The CLFS_MGMT_POLICY structure holds a description of a policy for managing a CLFS log.
CLFS_STREAM_ID_INFORMATION	The CLFS_STREAM_ID_INFORMATION structure holds a value that identifies a stream in a Common Log File System (CLFS) log.
CLS_CONTAINER_INFORMATION	The CLFS_CONTAINER_INFORMATION structure holds descriptive information for an individual container in a Common Log File System (CLFS) log.
CLS_INFORMATION	The CLFS_INFORMATION structure holds metadata and state information for a Common Log File System (CLFS) stream and/or its underlying physical log.
CLS_IO_STATISTICS	The CLFS_IO_STATISTICS structure holds I/O statistics data for a Common Log File System (CLFS) log.
CLS_IO_STATISTICS_HEADER	The CLFS_IO_STATISTICS_HEADER structure holds the header portion of a CLFS_IO_STATISTICS structure.
CLS_LSN	The CLFS_LSN structure identifies an individual record in a Common Log File System (CLFS) stream.
CLS_SCAN_CONTEXT	The CLFS_SCAN_CONTEXT structure holds context information to support a scan of the containers in a Common Log File System (CLFS) log.
CLS_WRITE_ENTRY	The CLFS_WRITE_ENTRY structure holds the address and size of a buffer that contains one unit of data to be written to a Common Log File System (CLFS) stream.

TITLE	DESCRIPTION
CM_EISA_FUNCTION_INFORMATION	The CM_EISA_FUNCTION_INFORMATION structure defines detailed EISA configuration information returned by HalGetBusData for the input BusDataType EisaConfiguration, or by HalGetBusDataByOffset for the input BusDataType EisaConfiguration and the Offset zero, assuming the caller-allocated Buffer is of sufficient Length.
CM_EISA_FUNCTION_INFORMATION	The CM_EISA_FUNCTION_INFORMATION structure defines detailed EISA configuration information returned by HalGetBusData for the input BusDataType EisaConfiguration, or by HalGetBusDataByOffset for the input BusDataType EisaConfiguration and the Offset zero, assuming the caller-allocated Buffer is of sufficient Length.
CM_EISA_SLOT_INFORMATION	The CM_EISA_SLOT_INFORMATION structure defines EISA configuration header information returned by HalGetBusData for the input BusDataType = EisaConfiguration, or by HalGetBusDataByOffset for the inputs BusDataType = EisaConfiguration and Offset = 0, assuming the caller-allocated Buffer is of sufficient Length.
CM_EISA_SLOT_INFORMATION	The CM_EISA_SLOT_INFORMATION structure defines EISA configuration header information returned by HalGetBusData for the input BusDataType = EisaConfiguration, or by HalGetBusDataByOffset for the inputs BusDataType = EisaConfiguration and Offset = 0, assuming the caller-allocated Buffer is of sufficient Length.
CM_FLOPPY_DEVICE_DATA	The CM_FLOPPY_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a floppy controller if the system can collect this information during the boot process.
CM_FULL_RESOURCE_DESCRIPTOR	The CM_FULL_RESOURCE_DESCRIPTOR structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.
CM_FULL_RESOURCE_DESCRIPTOR	The CM_FULL_RESOURCE_DESCRIPTOR structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.
CM_INT13_DRIVE_PARAMETER	The CM_INT13_DRIVE_PARAMETER structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a disk controller if the system can collect this information during the boot process.
CM_KEYBOARD_DEVICE_DATA	The CM_KEYBOARD_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a keyboard peripheral if the system can collect this information during the boot process.

TITLE	DESCRIPTION
CM_MCA_POS_DATA	The CM_MCA_POS_DATA structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.
CM_MCA_POS_DATA	The CM_MCA_POS_DATA structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.
CM_PARTIAL_RESOURCE_DESCRIPTOR	The CM_PARTIAL_RESOURCE_DESCRIPTOR structure specifies one or more system hardware resources, of a single type, assigned to a device.
CM_PARTIAL_RESOURCE_DESCRIPTOR	The CM_PARTIAL_RESOURCE_DESCRIPTOR structure specifies one or more system hardware resources, of a single type, assigned to a device.
CM_PARTIAL_RESOURCE_LIST	The CM_PARTIAL_RESOURCE_LIST structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.
CM_PARTIAL_RESOURCE_LIST	The CM_PARTIAL_RESOURCE_LIST structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.
CM_POWER_DATA	The CM_POWER_DATA structure contains information about a device's power management state and capabilities.
CM_POWER_DATA	The CM_POWER_DATA structure contains information about a device's power management state and capabilities.
CM_RESOURCE_LIST	The CM_RESOURCE_LIST structure specifies all of the system hardware resources assigned to a device.
CM_RESOURCE_LIST	The CM_RESOURCE_LIST structure specifies all of the system hardware resources assigned to a device.
CM_SCSI_DEVICE_DATA	The CM_SCSI_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a SCSI HBA if the system can collect this information during the boot process.
CM_SERIAL_DEVICE_DATA	The CM_SERIAL_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a serial controller if the system can collect this information during the boot process.
CONTROLLER_OBJECT	A controller object represents a hardware adapter or controller with homogenous devices that are the actual targets for I/O requests.
CORRELATION_VECTOR	Store the correlation vector that is used to reference events and the generated logs for diagnostic purposes.

TITLE	DESCRIPTION
COUNTED_REASON_CONTEXT	The COUNTED_REASON_CONTEXT structure contains one or more strings that give reasons for a power request.
COUNTED_REASON_CONTEXT	The COUNTED_REASON_CONTEXT structure contains one or more strings that give reasons for a power request.
D3COLD_SUPPORT_INTERFACE	The D3COLD_SUPPORT_INTERFACE interface structure contains pointers to the routines in the GUID_D3COLD_SUPPORT_INTERFACE driver interface.
DEVICE_CAPABILITIES	A DEVICE_CAPABILITIES structure describes PnP and power capabilities of a device. This structure is returned in response to an IRP_MN_QUERY_CAPABILITIES IRP.
DEVICE_DESCRIPTION	The DEVICE_DESCRIPTION structure describes the attributes of the physical device for which a driver is requesting a DMA adapter.
DEVICE_INTERFACE_CHANGE_NOTIFICATION	The DEVICE_INTERFACE_CHANGE_NOTIFICATION structure describes a device interface that has been enabled (arrived) or disabled (removed).
DEVICE_OBJECT	A device object represents a logical, virtual, or physical device for which a driver handles I/O requests.
DEVICE_RESET_INTERFACE_STANDARD	The DEVICE_RESET_INTERFACE_STANDARD structure enables function drivers to reset and recover malfunctioning devices. This structure describes the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
DMA_ADAPTER_INFO	The DMA_ADAPTER_INFO structure is a container for a DMA_ADAPTER_INFO_XXX structure that describes the capabilities of a system DMA controller.
DMA_ADAPTER_INFO_V1	The DMA_ADAPTER_INFO_V1 structure describes the capabilities of the system DMA controller that is represented by an adapter object.
DMA_TRANSFER_INFO	The DMA_TRANSFER_INFO structure is a container for a DMA_TRANSFER_INFO_XXX structure that describes the allocation requirements for a scatter/gather list.
DMA_TRANSFER_INFO_V1	The DMA_TRANSFER_INFO_V1 structure contains the allocation requirements for a scatter/gather list that describes the I/O data buffer for a DMA transfer.
DRIVER_OBJECT	Each driver object represents the image of a loaded kernel-mode driver.
ENLISTMENT_BASIC_INFORMATION	The ENLISTMENT_BASIC_INFORMATION structure contains information about an enlistment object.
EXT_DELETE_PARAMETERS	The EXT_DELETE_PARAMETERS structure contains an extended set of parameters for the ExDeleteTimer routine.

TITLE	DESCRIPTION
EXT_SET_PARAMETERS	The EXT_SET_PARAMETERS structure contains an extended set of parameters for the ExSetTimer routine.
FILE_ACCESS_INFORMATION	The FILE_ACCESS_INFORMATION structure is used to query for or set the access rights of a file.
FILE_ALIGNMENT_INFORMATION	The FILE_ALIGNMENT_INFORMATION structure is used as an argument to the ZwQueryInformationFile routine.
FILE_ALL_INFORMATION	The FILE_ALL_INFORMATION structure is a container for several FILE_XXX_INFORMATION structures.
FILE_ATTRIBUTE_TAG_INFORMATION	The FILE_ATTRIBUTE_TAG_INFORMATION structure is used as an argument to ZwQueryInformationFile.
FILE_BASIC_INFORMATION	The FILE_BASIC_INFORMATION structure is used as an argument to routines that query or set file information.
FILE_DISPOSITION_INFORMATION	The FILE_DISPOSITION_INFORMATION structure is used as an argument to the ZwSetInformationFile routine.
FILE_EA_INFORMATION	The FILE_EA_INFORMATION structure is used to query for the size of the extended attributes (EA) for a file.
FILE_END_OF_FILE_INFORMATION	The FILE_END_OF_FILE_INFORMATION structure is used as an argument to the ZwSetInformationFile routine.
FILE_FS_DEVICE_INFORMATION	The FILE_FS_DEVICE_INFORMATION structure provides file system device information about the type of device object associated with a file object.
FILE_FULL_EA_INFORMATION	The FILE_FULL_EA_INFORMATION structure provides extended attribute (EA) information. This structure is used primarily by network drivers.
FILE_IO_PRIORITY_HINT_INFORMATION	The FILE_IO_PRIORITY_HINT_INFORMATION structure is used by the ZwQueryInformationFile and ZwSetInformationFile routines to query and set the default IRP priority hint for requests on the specified file handle.
FILE_IS_REMOTE_DEVICE_INFORMATION	The FILE_IS_REMOTE_DEVICE_INFORMATION structure is used as an argument to the ZwQueryInformationFile routine.
FILE_MODE_INFORMATION	The FILE_MODE_INFORMATION structure is used to query or set the access mode of a file.
FILE_NAME_INFORMATION	The FILE_NAME_INFORMATION structure is used as argument to the ZwQueryInformationFile and ZwSetInformationFile routines.
FILE_NETWORK_OPEN_INFORMATION	The FILE_NETWORK_OPEN_INFORMATION structure is used as an argument to ZwQueryInformationFile.

TITLE	DESCRIPTION
FILE_OBJECT	The FILE_OBJECT structure is used by the system to represent a file object.
FILE_POSITION_INFORMATION	The FILE_POSITION_INFORMATION structure is used as an argument to routines that query or set file information.
FILE_STANDARD_INFORMATION	The FILE_STANDARD_INFORMATION structure is used as an argument to routines that query or set file information.
FILE_STANDARD_INFORMATION_EX	The FILE_STANDARD_INFORMATION_EX structure is used as an argument to routines that query or set file information.
FILE_VALID_DATA_LENGTH_INFORMATION	The FILE_VALID_DATA_LENGTH_INFORMATION structure is used as an argument to ZwSetInformationFile.
FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS	The FUNCTION_LEVEL_DEVICE_RESET_PARAMETER structure is used as an argument to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
GENERIC_MAPPING	The GENERIC_MAPPING structure describes the ACCESS_MASK value of specific access rights associated with each type of generic access right.
GROUP_AFFINITY	The GROUP_AFFINITY structure specifies a group number and the processor affinity within that group.
HARDWARE_COUNTER	The HARDWARE_COUNTER structure contains information about a hardware counter.
HWPROFILE_CHANGE_NOTIFICATION	The HWPROFILE_CHANGE_NOTIFICATION structure describes an event related to a hardware profile configuration change.
IMAGE_INFO	Used by driver's load-image routine (PLOAD_IMAGE_NOTIFY_ROUTINE) to specify image information.
IMAGE_INFO_EX	Extended version of the image information structure (see IMAGE_INFO).
IMAGE_POLICY_ENTRY	This structure is not supported.
IMAGE_POLICY_METADATA	This structure is not supported.
INTERFACE	The INTERFACE structure describes an interface that is exported by a driver for use by other drivers.
INTERFACE	The INTERFACE structure describes an interface that is exported by a driver for use by other drivers.
IO_CONNECT_INTERRUPT_PARAMETERS	The IO_CONNECT_INTERRUPT_PARAMETERS structure contains the parameters that a driver supplies to the IoConnectInterruptEx routine to register an interrupt service routine (ISR).

TITLE	DESCRIPTION
IO_DISCONNECT_INTERRUPT_PARAMETERS	The IO_DISCONNECT_INTERRUPT_PARAMETERS structure describes the parameters when unregistering an interrupt-handling routine with IoDisconnectInterruptEx.
IO_ERROR_LOG_PACKET	The IO_ERROR_LOG_PACKET structure serves as the header for an error log entry.
IO_INTERRUPT_MESSAGE_INFO	The IO_INTERRUPT_MESSAGE_INFO structure describes the driver's message-signaled interrupts.
IO_INTERRUPT_MESSAGE_INFO_ENTRY	The IO_INTERRUPT_MESSAGE_INFO_ENTRY structure describes the properties of a single message-signaled interrupt.
IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS	The IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure contains the connection context for a registered interrupt service routine (ISR) that was connected to an interrupt or interrupts by a previous call to the IoConnectInterruptEx routine.
IO_RESOURCE_DESCRIPTOR	The IO_RESOURCE_DESCRIPTOR structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of IO_RESOURCE_DESCRIPTOR structures is contained within each IO_RESOURCE_LIST structure.
IO_RESOURCE_DESCRIPTOR	The IO_RESOURCE_DESCRIPTOR structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of IO_RESOURCE_DESCRIPTOR structures is contained within each IO_RESOURCE_LIST structure.
IO_RESOURCE_LIST	The IO_RESOURCE_LIST structure describes a range of raw hardware resources, of various types, that can be used by a device.
IO_RESOURCE_LIST	The IO_RESOURCE_LIST structure describes a range of raw hardware resources, of various types, that can be used by a device.
IO_RESOURCE_REQUIREMENTS_LIST	The IO_RESOURCE_REQUIREMENTS_LIST structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.
IO_RESOURCE_REQUIREMENTS_LIST	The IO_RESOURCE_REQUIREMENTS_LIST structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.
IO_SECURITY_CONTEXT	The IO_SECURITY_CONTEXT structure represents the security context of an IRP_MJ_CREATE request.
IO_SESSION_CONNECT_INFO	The IO_SESSION_CONNECT_INFO structure provides information about a user session.

TITLE	DESCRIPTION
IO_SESSION_STATE_INFORMATION	The IO_SESSION_STATE_INFORMATION structure contains information about the state of a user session.
IO_SESSION_STATE_NOTIFICATION	The IO_SESSION_STATE_NOTIFICATION structure contains information that a kernel-mode driver supplies to the IoRegisterContainerNotification routine when the driver registers to receive notifications of session events.
IO_STACK_LOCATION	The IO_STACK_LOCATION structure defines an I/O stack location, which is an entry in the I/O stack that is associated with each IRP.
IO_STATUS_BLOCK	A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.
IRP	The IRP structure is a partially opaque structure that represents an I/O request packet. Drivers can use the following members of the IRP structure.
KBUGCHECK_ADD_PAGES	The KBUGCHECK_ADD_PAGES structure describes one or more pages of driver-supplied data to be written by a BugCheckAddPagesCallback callback routine to the crash dump file.
KBUGCHECK_DATA	The KBUGCHECK_DATA structure contains bug check parameters.
KBUGCHECK_DUMP_IO	The KBUGCHECK_DUMP_IO structure describes an I/O operation on the crash dump file.
KBUGCHECK_SECONDARY_DUMP_DATA	The KBUGCHECK_SECONDARY_DUMP_DATA structure describes a section of driver-supplied data to be written by BugCheckSecondaryDumpDataCallback to the crash dump file.
KDPC_WATCHDOG_INFORMATION	The KDPC_WATCHDOG_INFORMATION structure holds time-out information about the current deferred procedure call (DPC).
KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT	The KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure describes the notification context that is passed to a registered callback function when a new processor is dynamically added to a hardware partition.
KEY_BASIC_INFORMATION	The KEY_BASIC_INFORMATION structure defines a subset of the full information that is available for a registry key.
KEY_CACHED_INFORMATION	The KEY_CACHED_INFORMATION structure holds the cached information available for a registry key or subkey.
KEY_FULL_INFORMATION	The KEY_FULL_INFORMATION structure defines the information available for a registry key, including information about its subkeys and the maximum length for their names and value entries.

TITLE	DESCRIPTION
KEY_NAME_INFORMATION	The KEY_NAME_INFORMATION structure holds the name and name length of the key.
KEY_NODE_INFORMATION	The KEY_NODE_INFORMATION structure defines the basic information available for a registry (sub)key.
KEY_VALUE_BASIC_INFORMATION	The KEY_VALUE_BASIC_INFORMATION structure defines a subset of the full information available for a value entry of a registry key.
KEY_VALUE_ENTRY	The KEY_VALUE_ENTRY structure is used by the REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure to describe a single value entry for a registry key.
KEY_VALUE_FULL_INFORMATION	The KEY_VALUE_FULL_INFORMATION structure defines information available for a value entry of a registry key.
KEY_VALUE_PARTIAL_INFORMATION	The KEY_VALUE_PARTIAL_INFORMATION structure defines a subset of the value information available for a value entry of a registry key.
KEY_VIRTUALIZATION_INFORMATION	The KEY_VIRTUALIZATION_INFORMATION structure defines the basic information that is available for a registry key or subkey.
KEY_WRITE_TIME_INFORMATION	The KEY_WRITE_TIME_INFORMATION structure is used by the system to set the last write time for a registry key.
KTMOBJECT_CURSOR	The KTMOBJECT_CURSOR structure receives enumeration information about KTM objects when a component calls ZwEnumerateTransactionObject.
LINK_SHARE_ACCESS	The share access structure used by file systems for only link files.
MDL	An MDL structure is a partially opaque structure that represents a memory descriptor list (MDL).
MEMORY_BASIC_INFORMATION	Contains information about a range of pages in the virtual address space of a process.
MM_COPY_ADDRESS	The MM_COPY_ADDRESS structure contains either a virtual memory address or a physical memory address.
MM_PHYSICAL_ADDRESS_LIST	The MM_PHYSICAL_ADDRESS_LIST structure specifies a range of physical addresses.
OB_CALLBACK_REGISTRATION	The OB_CALLBACK_REGISTRATION structure specifies the parameters when the ObRegisterCallbacks routine registers ObjectPreCallback and ObjectPostCallback callback routines.
OB_OPERATION_REGISTRATION	The OB_OPERATION_REGISTRATION structure specifies ObjectPreCallback and ObjectPostCallback callback routines and the types of operations that the routines are called for.

TITLE	DESCRIPTION
OB_POST_CREATE_HANDLE_INFORMATION	The OB_POST_CREATE_HANDLE_INFORMATION structure provides information to a ObjectPostCallback routine about a thread or process handle that has been opened.
OB_POST_DUPLICATE_HANDLE_INFORMATION	The OB_POST_DUPLICATE_HANDLE_INFORMATION structure provides information to an ObjectPostCallback routine about a thread or process handle that has been duplicated.
OB_POST_OPERATION_INFORMATION	The OB_POST_OPERATION_INFORMATION structure provides information about a process or thread handle operation to an ObjectPostCallback routine.
OB_POST_OPERATION_PARAMETERS	The OB_POST_OPERATION_PARAMETERS union describes the operation-specific parameters for an ObjectPostCallback routine.
OB_PRE_CREATE_HANDLE_INFORMATION	The OB_PRE_CREATE_HANDLE_INFORMATION structure provides information to an ObjectPreCallback routine about a thread or process handle that is being opened.
OB_PRE_DUPLICATE_HANDLE_INFORMATION	The OB_PRE_DUPLICATE_HANDLE_INFORMATION structure provides information to an ObjectPreCallback routine about a thread or process handle that is being duplicated.
OB_PRE_OPERATION_INFORMATION	The OB_PRE_OPERATION_INFORMATION structure provides information about a process or thread handle operation to an ObjectPreCallback routine.
OB_PRE_OPERATION_PARAMETERS	The OB_PRE_OPERATION_PARAMETERS union describes the operation-specific parameters for an ObjectPreCallback routine.
OBJECT_ATTRIBUTES	The OBJECT_ATTRIBUTES structure specifies attributes that can be applied to objects or object handles by routines that create objects and/or return handles to objects.
OSVERSIONINFOEXW	The RTL_OSVERSIONINFOEXW structure contains operating system version information.
OSVERSIONINFOW	The RTL_OSVERSIONINFOW structure contains operating system version information.
PCI_COMMON_CONFIG	The PCI_COMMON_CONFIG structure is obsolete.
PCI_COMMON_CONFIG	The PCI_COMMON_CONFIG structure is obsolete.
PCI_MSIX_TABLE_CONFIG_INTERFACE	The PCI_MSIX_TABLE_CONFIG_INTERFACE structure enables device drivers to modify their MSI-X interrupt settings. This structure describes the GUID_MSIX_TABLE_CONFIG_INTERFACE interface.
PCI_SLOT_NUMBER	The PCI_SLOT_NUMBER structure is obsolete.
PCI_SLOT_NUMBER	The PCI_SLOT_NUMBER structure is obsolete.

TITLE	DESCRIPTION
PEP_ABANDON_DEVICE	The PEP_ABANDON_DEVICE structure identifies a device that has been abandoned and will no longer be used by the operating system.
PEP_ABANDON_DEVICE	The PEP_ABANDON_DEVICE structure identifies a device that has been abandoned and will no longer be used by the operating system.
PEP_ACPI_ABANDON_DEVICE	The PEP_ACPI_ABANDON_DEVICE structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.
PEP_ACPI_ABANDON_DEVICE	The PEP_ACPI_ABANDON_DEVICE structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE	The PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure contains an enumeration of the objects in the namespace of the device.
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE	The PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure contains an enumeration of the objects in the namespace of the device.
PEP_ACPI_EVALUATE_CONTROL_METHOD	The PEP_ACPI_EVALUATE_CONTROL_METHOD structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.
PEP_ACPI_EVALUATE_CONTROL_METHOD	The PEP_ACPI_EVALUATE_CONTROL_METHOD structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.
PEP_ACPI_EXTENDED_ADDRESS	The PEP_ACPI_EXTENDED_ADDRESS structure is used to report resource usage in the address space such as memory and IO.
PEP_ACPI_EXTENDED_ADDRESS	The PEP_ACPI_EXTENDED_ADDRESS structure is used to report resource usage in the address space such as memory and IO.
PEP_ACPI_GPIO_RESOURCE	The PEP_ACPI_GPIO_RESOURCE structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.
PEP_ACPI_GPIO_RESOURCE	The PEP_ACPI_GPIO_RESOURCE structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.
PEP_ACPI_INTERRUPT_RESOURCE	The PEP_ACPI_INTERRUPT_RESOURCE structure describes an ACPI interrupt resource.
PEP_ACPI_INTERRUPT_RESOURCE	The PEP_ACPI_INTERRUPT_RESOURCE structure describes an ACPI interrupt resource.

TITLE	DESCRIPTION
PEP_ACPI_IO_MEMORY_RESOURCE	The PEP_ACPI_IO_MEMORY_RESOURCE structure describes an ACPI IO port descriptor resource.
PEP_ACPI_IO_MEMORY_RESOURCE	The PEP_ACPI_IO_MEMORY_RESOURCE structure describes an ACPI IO port descriptor resource.
PEP_ACPI_OBJECT_NAME	The PEP_ACPI_OBJECT_NAME union contains the four-character name of an ACPI object.
PEP_ACPI_OBJECT_NAME	The PEP_ACPI_OBJECT_NAME union contains the four-character name of an ACPI object.
PEP_ACPI_OBJECT_NAME_WITH_TYPE	The PEP_ACPI_OBJECT_NAME_WITH_TYPE structure that specifies both the path-relative name of an ACPI object and the type of this object.
PEP_ACPI_OBJECT_NAME_WITH_TYPE	The PEP_ACPI_OBJECT_NAME_WITH_TYPE structure that specifies both the path-relative name of an ACPI object and the type of this object.
PEP_ACPI_PREPARE_DEVICE	The PEP_ACPI_PREPARE_DEVICE structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.
PEP_ACPI_PREPARE_DEVICE	The PEP_ACPI_PREPARE_DEVICE structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure contains a list of raw resources that are needed to control power to the device.
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure contains a list of raw resources that are needed to control power to the device.
PEP_ACPI_QUERY_OBJECT_INFORMATION	The PEP_ACPI_QUERY_OBJECT_INFORMATION structure contains information about an ACPI object.
PEP_ACPI_QUERY_OBJECT_INFORMATION	The PEP_ACPI_QUERY_OBJECT_INFORMATION structure contains information about an ACPI object.
PEP_ACPI_REGISTER_DEVICE	The PEP_ACPI_REGISTER_DEVICE structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.
PEP_ACPI_REGISTER_DEVICE	The PEP_ACPI_REGISTER_DEVICE structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.
PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES	The PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.

TITLE	DESCRIPTION
PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES	The PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.
PEP_ACPI_RESOURCE	The PEP_ACPI_RESOURCE structure contains hardware details for a specific ACPI resource.
PEP_ACPI_RESOURCE	The PEP_ACPI_RESOURCE structure contains hardware details for a specific ACPI resource.
PEP_ACPI_RESOURCE_FLAGS	The PEP_ACPI_RESOURCE_FLAGS structure contains flags describing an ACPI resource.
PEP_ACPI_RESOURCE_FLAGS	The PEP_ACPI_RESOURCE_FLAGS structure contains flags describing an ACPI resource.
PEP_ACPI_SPB_I2C_RESOURCE	The PEP_ACPI_SPB_I2C_RESOURCE structure describes an ACPI I2C serial bus resource.
PEP_ACPI_SPB_I2C_RESOURCE	The PEP_ACPI_SPB_I2C_RESOURCE structure describes an ACPI I2C serial bus resource.
PEP_ACPI_SPB_RESOURCE	The PEP_ACPI_SPB_RESOURCE structure describes an ACPI serial bus connection resource.
PEP_ACPI_SPB_RESOURCE	The PEP_ACPI_SPB_RESOURCE structure describes an ACPI serial bus connection resource.
PEP_ACPI_SPB_SPI_RESOURCE	The PEP_ACPI_SPB_SPI_RESOURCE structure describes an ACPI SPI serial bus resource.
PEP_ACPI_SPB_SPI_RESOURCE	The PEP_ACPI_SPB_SPI_RESOURCE structure describes an ACPI SPI serial bus resource.
PEP_ACPI_SPB_UART_RESOURCE	The PEP_ACPI_SPB_UART_RESOURCE structure describes an ACPI UART serial bus resource.
PEP_ACPI_SPB_UART_RESOURCE	The PEP_ACPI_SPB_UART_RESOURCE structure describes an ACPI UART serial bus resource.
PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.
PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.
PEP_ACPI_UNREGISTER_DEVICE	The PEP_ACPI_UNREGISTER_DEVICE structure contains information about a device that has been unregistered from ACPI services.

TITLE	DESCRIPTION
PEP_ACPI_UNREGISTER_DEVICE	The PEP_ACPI_UNREGISTER_DEVICE structure contains information about a device that has been unregistered from ACPI services.
PEP_COMPONENT_ACTIVE	The PEP_COMPONENT_ACTIVE structure identifies a component that is making a transition between the idle condition and the active condition.
PEP_COMPONENT_PERF_INFO	The PEP_COMPONENT_PERF_INFO structure describes the performance states (P-states) of a component.
PEP_COMPONENT_PERF_INFO	The PEP_COMPONENT_PERF_INFO structure describes the performance states (P-states) of a component.
PEP_COMPONENT_PERF_SET	The PEP_COMPONENT_PERF_SET structure describes the performance states (P-states) in a P-state set.
PEP_COMPONENT_PERF_SET	The PEP_COMPONENT_PERF_SET structure describes the performance states (P-states) in a P-state set.
PEP_COMPONENT_PERF_STATE_REQUEST	The PEP_COMPONENT_PERF_STATE_REQUEST structure specifies a performance state (P-state) set and a new performance level to assign to this set.
PEP_COMPONENT_PERF_STATE_REQUEST	The PEP_COMPONENT_PERF_STATE_REQUEST structure specifies a performance state (P-state) set and a new performance level to assign to this set.
PEP_COMPONENT_PLATFORM_CONSTRAINTS	The PEP_COMPONENT_PLATFORM_CONSTRAINTS structure describes the lowest-powered Fx state of that a component can be in when the platform is in a particular idle state.
PEP_COMPONENT_PLATFORM_CONSTRAINTS	The PEP_COMPONENT_PLATFORM_CONSTRAINTS structure describes the lowest-powered Fx state of that a component can be in when the platform is in a particular idle state.
PEP_COMPONENT_V2	The PEP_COMPONENT_V2 structure specifies the power state attributes of a component in the device.
PEP_COMPONENT_V2	The PEP_COMPONENT_V2 structure specifies the power state attributes of a component in the device.
PEP_COORDINATED_DEPENDENCY_OPTION	The PEP_COORIDNATED_DEPENDENCY_OPTION structure describes a coordinated idle state's dependency to the OS.
PEP_COORDINATED_DEPENDENCY_OPTION	The PEP_COORIDNATED_DEPENDENCY_OPTION structure describes a coordinated idle state's dependency to the OS.
PEP_COORDINATED_IDLE_STATE	The PEP_COORIDNATED_IDLE_STATE structure describes a coordinated idle state to the OS.
PEP_COORDINATED_IDLE_STATE	The PEP_COORIDNATED_IDLE_STATE structure describes a coordinated idle state to the OS.

TITLE	DESCRIPTION
PEP_CRASHDUMP_INFORMATION	The PEP_CRASHDUMP_INFORMATION structure contains information about a crash-dump device.
PEP_CRASHDUMP_INFORMATION	The PEP_CRASHDUMP_INFORMATION structure contains information about a crash-dump device.
PEP_DEBUGGER_TRANSITION_REQUIREMENTS	The PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure indicates the platform idle states for which the debugger device must be turned on.
PEP_DEBUGGER_TRANSITION_REQUIREMENTS	The PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure indicates the platform idle states for which the debugger device must be turned on.
PEP_DEVICE_PLATFORM_CONSTRAINTS	The PEP_DEVICE_PLATFORM_CONSTRAINTS structure specifies the constraints for entry to the various Dx power states that are supported by a device.
PEP_DEVICE_PLATFORM_CONSTRAINTS	The PEP_DEVICE_PLATFORM_CONSTRAINTS structure specifies the constraints for entry to the various Dx power states that are supported by a device.
PEP_DEVICE_POWER_STATE	The PEP_DEVICE_POWER_STATE structure indicates the status of a transition to a new Dx (device power) state.
PEP_DEVICE_POWER_STATE	The PEP_DEVICE_POWER_STATE structure indicates the status of a transition to a new Dx (device power) state.
PEP_DEVICE_REGISTER_V2	The PEP_DEVICE_REGISTER structure describes all the components in a particular device.
PEP_DEVICE_REGISTER_V2	The PEP_DEVICE_REGISTER structure describes all the components in a particular device.
PEP_DEVICE_STARTED	The PEP_DEVICE_STARTED structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).
PEP_DEVICE_STARTED	The PEP_DEVICE_STARTED structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).
PEP_INFORMATION	The PEP_INFORMATION structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).
PEP_INFORMATION	The PEP_INFORMATION structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).

TITLE	DESCRIPTION
PEP_KERNEL_INFORMATION_STRUCT_V1	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V2	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V3	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V3	The PEP_KERNEL_INFORMATION_STRUCT_V3 structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_LOW_POWER_EPOCH	The PEP_LOW_POWER_EPOCH structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).
PEP_LOW_POWER_EPOCH	The PEP_LOW_POWER_EPOCH structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).
PEP_NOTIFY_COMPONENT_IDLE_STATE	The PEP_NOTIFY_COMPONENT_IDLE_STATE structure contains status information about a component's pending transition to a new Fx power state.
PEP_NOTIFY_COMPONENT_IDLE_STATE	The PEP_NOTIFY_COMPONENT_IDLE_STATE structure contains status information about a component's pending transition to a new Fx power state.
PEP_PERF_STATE	The PEP_PERF_STATE structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.
PEP_PERF_STATE	The PEP_PERF_STATE structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.
PEP_PLATFORM_IDLE_STATE	The PEP_PLATFORM_IDLE_STATE structure specifies the properties of a platform idle state.
PEP_PLATFORM_IDLE_STATE	The PEP_PLATFORM_IDLE_STATE structure specifies the properties of a platform idle state.
PEP_PLATFORM_IDLE_STATE_UPDATE	The PEP_PLATFORM_IDLE_STATE_UPDATE structure contains the updated properties of a platform idle state.
PEP_PLATFORM_IDLE_STATE_UPDATE	The PEP_PLATFORM_IDLE_STATE_UPDATE structure contains the updated properties of a platform idle state.

TITLE	DESCRIPTION
PEP_POWER_CONTROL_COMPLETE	The PEP_POWER_CONTROL_COMPLETE structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.
PEP_POWER_CONTROL_COMPLETE	The PEP_POWER_CONTROL_COMPLETE structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.
PEP_POWER_CONTROL_REQUEST	The PEP_POWER_CONTROL_REQUEST structure contains a request from a driver for a power control operation.
PEP_POWER_CONTROL_REQUEST	The PEP_POWER_CONTROL_REQUEST structure contains a request from a driver for a power control operation.
PEP_PPM_CONTEXT_QUERY_PARKING_PAGE	The PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure describes the parking page for a processor.
PEP_PPM_CONTEXT_QUERY_PARKING_PAGE	The PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure describes the parking page for a processor.
PEP_PPM_CST_STATE	The PEP_PPM_CST_STATE structure specifies the properties of a C state (ACPI processor power state).
PEP_PPM_CST_STATE	The PEP_PPM_CST_STATE structure specifies the properties of a C state (ACPI processor power state).
PEP_PPM_CST_STATES	The PEP_PPM_CST_STATES structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.
PEP_PPM_CST_STATES	The PEP_PPM_CST_STATES structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.
PEP_PPM_ENTER_SYSTEM_STATE	Used in the PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE notification to notify PEP that the system is about to enter a system power state. .
PEP_PPM_ENTER_SYSTEM_STATE	Used in the PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE notification to notify PEP that the system is about to enter a system power state. .
PEP_PPM_FEEDBACK_READ	The PEP_PPM_FEEDBACK_READ structure contains the value read from a processor performance feedback counter.
PEP_PPM_FEEDBACK_READ	The PEP_PPM_FEEDBACK_READ structure contains the value read from a processor performance feedback counter.
PEP_PPM_IDLE_CANCEL	The PEP_PPM_IDLE_CANCEL structure indicates why the processor could not enter the previously selected idle state.

TITLE	DESCRIPTION
PEP_PPM_IDLE_COMPLETE	The PEP_PPM_IDLE_COMPLETE structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_COMPLETE	The PEP_PPM_IDLE_COMPLETE structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_COMPLETE_V2	The PEP_PPM_IDLE_COMPLETE_V2 structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_COMPLETE_V2	The PEP_PPM_IDLE_COMPLETE_V2 structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_EXECUTE	The PEP_PPM_IDLE_EXECUTE structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_EXECUTE	The PEP_PPM_IDLE_EXECUTE structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_EXECUTE_V2	The PEP_PPM_IDLE_EXECUTE_V2 structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_EXECUTE_V2	The PEP_PPM_IDLE_EXECUTE_V2 structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_SELECT	The PEP_PPM_IDLE_SELECT structure describes the most energy-efficient idle state that the processor can enter and still satisfy the constraints specified by the operating system.
PEP_PPM_INITIATE_WAKE	The PEP_PPM_INITIATE_WAKE structure indicates whether a processor requires an interrupt to wake up from an idle state.
PEP_PPM_INITIATE_WAKE	The PEP_PPM_INITIATE_WAKE structure indicates whether a processor requires an interrupt to wake up from an idle state.
PEP_PPM_IS_PROCESSOR_HALTED	The PEP_PPM_IS_PROCESSOR_HALTED structure indicates whether the processor is currently halted in its selected idle state.
PEP_PPM_IS_PROCESSOR_HALTED	The PEP_PPM_IS_PROCESSOR_HALTED structure indicates whether the processor is currently halted in its selected idle state.
PEP_PPM_LPI_COMPLETE	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_PARK_MASK	The PEP_PROCESSOR_PARK_MASK structure contains the current core parking mask.

TITLE	DESCRIPTION
PEP_PPM_PARK_MASK	The PEP_PROCESSOR_PARK_MASK structure contains the current core parking mask.
PEP_PPM_PARK_SELECTION	The PEP_PPM_PARK_SELECTION structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PARK_SELECTION	The PEP_PPM_PARK_SELECTION structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PARK_SELECTION_V2	The PEP_PPM_PARK_SELECTION_V2 structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PARK_SELECTION_V2	The PEP_PPM_PARK_SELECTION_V2 structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PERF_CHECK_COMPLETE	The PEP_PPM_PERF_CHECK_COMPLETE structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.
PEP_PPM_PERF_CHECK_COMPLETE	The PEP_PPM_PERF_CHECK_COMPLETE structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.
PEP_PPM_PERF_CONSTRAINTS	The PEP_PPM_PERF_CONSTRAINTS structure describes the performance limits to apply to the processor.
PEP_PPM_PERF_CONSTRAINTS	The PEP_PPM_PERF_CONSTRAINTS structure describes the performance limits to apply to the processor.
PEP_PPM_PERF_SET	The PEP_PPM_PERF_SET structure specifies the new performance level that the operating system is requesting for the processor.
PEP_PPM_PERF_SET	The PEP_PPM_PERF_SET structure specifies the new performance level that the operating system is requesting for the processor.
PEP_PPM_PERF_SET_STATE	Used in the PEP_NOTIFY_PPM_PERF_SET notification at runtime to set the current operating performance of the processor. .
PEP_PPM_PERF_SET_STATE	Used in the PEP_NOTIFY_PPM_PERF_SET notification at runtime to set the current operating performance of the processor. .

TITLE	DESCRIPTION
PEP_PPM_PLATFORM_STATE_RESIDENCIES	The PEP_PPM_PLATFORM_STATE_RESIDENCIES structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.
PEP_PPM_PLATFORM_STATE_RESIDENCIES	The PEP_PPM_PLATFORM_STATE_RESIDENCIES structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.
PEP_PPM_PLATFORM_STATE_RESIDENCY	The PEP_PPM_PLATFORM_STATE_RESIDENCY structure specifies the accumulated residency time and transition count for a particular platform idle state.
PEP_PPM_PLATFORM_STATE_RESIDENCY	The PEP_PPM_PLATFORM_STATE_RESIDENCY structure specifies the accumulated residency time and transition count for a particular platform idle state.
PEP_PPM_QUERY_CAPABILITIES	The PEP_PPM_QUERY_CAPABILITIES structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).
PEP_PPM_QUERY_CAPABILITIES	The PEP_PPM_QUERY_CAPABILITIES structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).
PEP_PPM_QUERY_COORDINATED_DEPENDENCY	The PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure describes dependencies for coordinated idle states.
PEP_PPM_QUERY_COORDINATED_DEPENDENCY	The PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure describes dependencies for coordinated idle states.
PEP_PPM_QUERY_COORDINATED_STATES	The PEP_PPM_QUERY_COORDINATED_STATES structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.
PEP_PPM_QUERY_COORDINATED_STATES	The PEP_PPM_QUERY_COORDINATED_STATES structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.
PEP_PPM_QUERY_DISCRETE_PERF_STATES	Used in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification that stores the list of discrete performance states that PEP supports, if the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification indicates support for discrete performance states. .
PEP_PPM_QUERY_DISCRETE_PERF_STATES	Used in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification that stores the list of discrete performance states that PEP supports, if the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification indicates support for discrete performance states. .
PEP_PPM_QUERY_DOMAIN_INFO	Used in the PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO notification that queries for information about a performance domain. .

TITLE	DESCRIPTION
PEP_PPM_QUERY_DOMAIN_INFO	Used in the PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO notification that queries for information about a performance domain. .
PEP_PPM_QUERY_FEEDBACK_COUNTERS	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_QUERY_FEEDBACK_COUNTERS	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_QUERY_IDLE_STATES	The PEP_PPM_QUERY_IDLE_STATES structure describes the idle states of a particular processor.
PEP_PPM_QUERY_IDLE_STATES_V2	The PEP_PPM_QUERY_IDLE_STATES_V2 structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.
PEP_PPM_QUERY_IDLE_STATES_V2	The PEP_PPM_QUERY_IDLE_STATES_V2 structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.
PEP_PPM_QUERY_LP_SETTINGS	The PEP_PPM_QUERY_LP_SETTINGS structure contains a kernel handle to the registry key that contains the power optimization settings that the platform extension plug-in (PEP) has defined for each power scenario.
PEP_PPM_QUERY_PERF_CAPABILITIES	The PEP_PPM_QUERY_PERF_CAPABILITIES structure describes the performance capabilities of the processors in the specified processor performance domain.
PEP_PPM_QUERY_PERF_CAPABILITIES	The PEP_PPM_QUERY_PERF_CAPABILITIES structure describes the performance capabilities of the processors in the specified processor performance domain.
PEP_PPM_QUERY_PLATFORM_STATE	The PEP_PPM_QUERY_PLATFORM_STATE structure contains information about a platform idle state.
PEP_PPM_QUERY_PLATFORM_STATE	The PEP_PPM_QUERY_PLATFORM_STATE structure contains information about a platform idle state.
PEP_PPM_QUERY_PLATFORM_STATES	The PEP_PPM_QUERY_PLATFORM_STATES structure specifies the number of platform idle states the hardware platform supports.
PEP_PPM_QUERY_PLATFORM_STATES	The PEP_PPM_QUERY_PLATFORM_STATES structure specifies the number of platform idle states the hardware platform supports.

TITLE	DESCRIPTION
PEP_PPM_QUERY_STATE_NAME	The PEP_PPM_QUERY_STATE_NAME structure contains information about a specific coordinated or platform idle state.
PEP_PPM_QUERY_STATE_NAME	The PEP_PPM_QUERY_STATE_NAME structure contains information about a specific coordinated or platform idle state.
PEP_PPM_QUERY_VETO_REASON	The PEP_PPM_QUERY_VETO_REASON structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.
PEP_PPM_QUERY_VETO_REASON	The PEP_PPM_QUERY_VETO_REASON structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.
PEP_PPM_QUERY_VETO_REASONS	The PEP_PPM_QUERY_VETO_REASONS structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.
PEP_PPM_QUERY_VETO_REASONS	The PEP_PPM_QUERY_VETO_REASONS structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.
PEP_PPM_RESUME_FROM_SYSTEM_STATE	Used by the PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE notification that notifies the PEP that the system has just resumed from a system power state.
PEP_PPM_RESUME_FROM_SYSTEM_STATE	Used by the PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE notification that notifies the PEP that the system has just resumed from a system power state.
PEP_PPM_TEST_IDLE_STATE	The PEP_PPM_TEST_IDLE_STATE structure contains information about whether the processor can immediately enter a processor idle state.
PEP_PPM_TEST_IDLE_STATE	The PEP_PPM_TEST_IDLE_STATE structure contains information about whether the processor can immediately enter a processor idle state.
PEP_PREPARE_DEVICE	The PEP_PREPARE_DEVICE structure identifies a device that must be started up in preparation for its use by the operating system.
PEP_PREPARE_DEVICE	The PEP_PREPARE_DEVICE structure identifies a device that must be started up in preparation for its use by the operating system.
PEP_PROCESSOR_FEEDBACK_COUNTER	The PEP_PROCESSOR_FEEDBACK_COUNTER structure describes a feedback counter to the operating system.
PEP_PROCESSOR_FEEDBACK_COUNTER	The PEP_PROCESSOR_FEEDBACK_COUNTER structure describes a feedback counter to the operating system.

TITLE	DESCRIPTION
PEP_PROCESSOR_IDLE_CONSTRAINTS	The PEP_PROCESSOR_IDLE_CONSTRAINTS structure specifies a set of constraints that the PEP uses to select a processor idle state.
PEP_PROCESSOR_IDLE_DEPENDENCY	The PEP_PROCESSOR_IDLE_DEPENDENCY structure specifies the dependencies of a platform idle state on the specified processor.
PEP_PROCESSOR_IDLE_DEPENDENCY	The PEP_PROCESSOR_IDLE_DEPENDENCY structure specifies the dependencies of a platform idle state on the specified processor.
PEP_PROCESSOR_IDLE_STATE	The PEP_PROCESSOR_IDLE_STATE structure describes the capabilities of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_UPDATE	The PEP_PROCESSOR_IDLE_STATE_UPDATE structure contains the updated properties of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_UPDATE	The PEP_PROCESSOR_IDLE_STATE_UPDATE structure contains the updated properties of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_V2	The PEP_PROCESSOR_IDLE_STATE_V2 structure describes a processor idle state that the platform extension plug-in (PEP) supports.
PEP_PROCESSOR_IDLE_STATE_V2	The PEP_PROCESSOR_IDLE_STATE_V2 structure describes a processor idle state that the platform extension plug-in (PEP) supports.
PEP_PROCESSOR_PARK_PREFERENCE	The PEP_PROCESSOR_PARK_PREFERENCE structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.
PEP_PROCESSOR_PARK_PREFERENCE	The PEP_PROCESSOR_PARK_PREFERENCE structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.
PEP_PROCESSOR_PARK_STATE	The PEP_PROCESSOR_PARK_STATE structure describes the parking state for a single processor.
PEP_PROCESSOR_PARK_STATE	The PEP_PROCESSOR_PARK_STATE structure describes the parking state for a single processor.
PEP_PROCESSOR_PERF_STATE	Use in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification. This structure describes the properties of a single performance state. .
PEP_PROCESSOR_PERF_STATE	Use in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification. This structure describes the properties of a single performance state. .

TITLE	DESCRIPTION
PEP_QUERY_COMPONENT_PERF_CAPABILITIES	The PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure specifies the number of performance state (P-state) sets that are defined for a component.
PEP_QUERY_COMPONENT_PERF_CAPABILITIES	The PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure specifies the number of performance state (P-state) sets that are defined for a component.
PEP_QUERY_COMPONENT_PERF_SET	The PEP_QUERY_COMPONENT_PERF_SET structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_SET	The PEP_QUERY_COMPONENT_PERF_SET structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_SET_NAME	The PEP_QUERY_COMPONENT_PERF_SET_NAME structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_SET_NAME	The PEP_QUERY_COMPONENT_PERF_SET_NAME structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_STATES	The PEP_QUERY_COMPONENT_PERF_STATES structure contains a list of discrete performance state (P-state) values for the specified P-state set.
PEP_QUERY_COMPONENT_PERF_STATES	The PEP_QUERY_COMPONENT_PERF_STATES structure contains a list of discrete performance state (P-state) values for the specified P-state set.
PEP_QUERY_CURRENT_COMPONENT_PERF_STATE	The PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure contains information about the current P-state in the specified P-state set.
PEP_QUERY_CURRENT_COMPONENT_PERF_STATE	The PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure contains information about the current P-state in the specified P-state set.
PEP_QUERY_SOC_SUBSYSTEM	The PEP_QUERY_SOC_SUBSYSTEM structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM	The PEP_QUERY_SOC_SUBSYSTEM structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME	The PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.

TITLE	DESCRIPTION
PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME	The PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM_COUNT	The PEP_QUERY_SOC_SUBSYSTEM_COUNT structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.
PEP_QUERY_SOC_SUBSYSTEM_COUNT	The PEP_QUERY_SOC_SUBSYSTEM_COUNT structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.
PEP_QUERY_SOC_SUBSYSTEM_METADATA	The PEP_QUERY_SOC_SUBSYSTEM_METADATA structure is used with the PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.
PEP_QUERY_SOC_SUBSYSTEM_METADATA	The PEP_QUERY_SOC_SUBSYSTEM_METADATA structure is used with the PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.
PEP_REGISTER_COMPONENT_PERF_STATES	The PEP_REGISTER_COMPONENT_PERF_STATES structure describes the performance states (P-states) of the specified component.
PEP_REGISTER_COMPONENT_PERF_STATES	The PEP_REGISTER_COMPONENT_PERF_STATES structure describes the performance states (P-states) of the specified component.
PEP_REGISTER_CRASHDUMP_DEVICE	The PEP_REGISTER_CRASHDUMP_DEVICE structure provides a callback routine to turn on a crash-dump device.
PEP_REGISTER_CRASHDUMP_DEVICE	The PEP_REGISTER_CRASHDUMP_DEVICE structure provides a callback routine to turn on a crash-dump device.
PEP_REGISTER_DEBUGGER	The PEP_REGISTER_DEBUGGER structure identifies a registered device that is a core system resource that provides debugger transport.
PEP_REGISTER_DEBUGGER	The PEP_REGISTER_DEBUGGER structure identifies a registered device that is a core system resource that provides debugger transport.
PEP_REGISTER_DEVICE_V2	The PEP_REGISTER_DEVICE_V2 structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).
PEP_REGISTER_DEVICE_V2	The PEP_REGISTER_DEVICE_V2 structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).

TITLE	DESCRIPTION
PEP_REQUEST_COMPONENT_PERF_STATE	The PEP_REQUEST_COMPONENT_PERF_STATE structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).
PEP_REQUEST_COMPONENT_PERF_STATE	The PEP_REQUEST_COMPONENT_PERF_STATE structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).
PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING	The PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.
PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING	The PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.
PEP_SOC_SUBSYSTEM_METADATA	The PEP_SOC_SUBSYSTEM_METADATA structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).
PEP_SOC_SUBSYSTEM_METADATA	The PEP_SOC_SUBSYSTEM_METADATA structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).
PEP_SYSTEM_LATENCY	The PEP_SYSTEM_LATENCY structure specifies the new value for the system latency tolerance.
PEP_SYSTEM_LATENCY	The PEP_SYSTEM_LATENCY structure specifies the new value for the system latency tolerance.
PEP_UNMASKED_INTERRUPT_FLAGS	The PEP_UNMASKED_INTERRUPT_FLAGS union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.
PEP_UNMASKED_INTERRUPT_FLAGS	The PEP_UNMASKED_INTERRUPT_FLAGS union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.
PEP_UNMASKED_INTERRUPT_INFORMATION	The PEP_UNMASKED_INTERRUPT_INFORMATION structure contains information about an interrupt source.
PEP_UNMASKED_INTERRUPT_INFORMATION	The PEP_UNMASKED_INTERRUPT_INFORMATION structure contains information about an interrupt source.

TITLE	DESCRIPTION
PEP_UNREGISTER_DEVICE	The PEP_UNREGISTER_DEVICE structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).
PEP_UNREGISTER_DEVICE	The PEP_UNREGISTER_DEVICE structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).
PEP_WORK	The PEP_WORK structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).
PEP_WORK	The PEP_WORK structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).
PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE	The PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).
PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE	The PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).
PEP_WORK_ACPI_NOTIFY	The PEP_WORK_ACPI_NOTIFY structure contains the ACPI Notify code for a device that has generated a hardware event.
PEP_WORK_ACPI_NOTIFY	The PEP_WORK_ACPI_NOTIFY structure contains the ACPI Notify code for a device that has generated a hardware event.
PEP_WORK_ACTIVE_COMPLETE	The PEP_WORK_ACTIVE_COMPLETE structure identifies a component that is now in the active condition.
PEP_WORK_COMPLETE_IDLE_STATE	The PEP_WORK_COMPLETE_IDLE_STATE structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new Fx power state.
PEP_WORK_COMPLETE_IDLE_STATE	The PEP_WORK_COMPLETE_IDLE_STATE structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new Fx power state.
PEP_WORK_COMPLETE_PERF_STATE	The PEP_WORK_COMPLETE_PERF_STATE structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.
PEP_WORK_COMPLETE_PERF_STATE	The PEP_WORK_COMPLETE_PERF_STATE structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.

TITLE	DESCRIPTION
PEP_WORK_DEVICE_IDLE	The PEP_WORK_DEVICE_IDLE structure indicates whether to ignore the idle time-out for the specified device.
PEP_WORK_DEVICE_POWER	The PEP_WORK_DEVICE_POWER structure describes the new power requirements for the specified device.
PEP_WORK_IDLE_STATE	The PEP_WORK_IDLE_STATE structure contains a request to transition a component to an Fx power state.
PEP_WORK_INFORMATION	The PEP_WORK_INFORMATION structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).
PEP_WORK_INFORMATION	The PEP_WORK_INFORMATION structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).
PEP_WORK_POWER_CONTROL	The PEP_WORK_POWER_CONTROL structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.
PEP_WORK_POWER_CONTROL	The PEP_WORK_POWER_CONTROL structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.
PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR	The PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure describes the counter resources available on the platform.
PHYSICAL_COUNTER_RESOURCE_LIST	The PHYSICAL_COUNTER_RESOURCE_LIST structure describes an array of PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structures.
PLUGPLAY_NOTIFICATION_HEADER	A PLUGPLAY_NOTIFICATION_HEADER structure is included at the beginning of each PnP notification structure, such as a DEVICE_INTERFACE_CHANGE_NOTIFICATION structure.
PNP_BUS_INFORMATION	The PNP_BUS_INFORMATION structure describes a bus.
PNP_LOCATION_INTERFACE	The PNP_LOCATION_INTERFACE structure describes the GUID_PNP_LOCATION_INTERFACE interface.
PO_FX_COMPONENT_IDLE_STATE	The PO_FX_COMPONENT_IDLE_STATE structure specifies the attributes of an Fx power state of a component in a device.
PO_FX_COMPONENT_IDLE_STATE	The PO_FX_COMPONENT_IDLE_STATE structure specifies the attributes of an Fx power state of a component in a device.
PO_FX_COMPONENT_PERF_INFO	The PO_FX_COMPONENT_PERF_INFO structure describes all the sets of performance states for a single component within a device.
PO_FX_COMPONENT_PERF_INFO	The PO_FX_COMPONENT_PERF_INFO structure describes all the sets of performance states for a single component within a device.

TITLE	DESCRIPTION
PO_FX_COMPONENT_PERF_SET	The PO_FX_COMPONENT_PERF_SET structure represents a set of performance states for a single component within a device.
PO_FX_COMPONENT_PERF_SET	The PO_FX_COMPONENT_PERF_SET structure represents a set of performance states for a single component within a device.
PO_FX_COMPONENT_V1	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_V1	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_V2	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_V2	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_CORE_DEVICE	The PO_FX_CORE_DEVICE structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.
PO_FX_CORE_DEVICE	The PO_FX_CORE_DEVICE structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.
PO_FX_DEVICE_V1	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_DEVICE_V1	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_DEVICE_V2	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_DEVICE_V2	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_PERF_STATE	The PO_FX_PERF_STATE structure represents a performance state for a single component within a device.
PO_FX_PERF_STATE	The PO_FX_PERF_STATE structure represents a performance state for a single component within a device.
PO_FX_PERF_STATE_CHANGE	The PO_FX_PERF_STATE_CHANGE structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.

TITLE	DESCRIPTION
PO_FX_PERF_STATE_CHANGE	The PO_FX_PERF_STATE_CHANGE structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.
PO_SPR_ACTIVE_SESSION_DATA	Allow drivers to register/unregister for SPR Active session start and end notifications.
POWER_PLATFORM_INFORMATION	The POWER_PLATFORM_INFORMATION structure contains information about the power capabilities of the system.
POWER_PLATFORM_INFORMATION	The POWER_PLATFORM_INFORMATION structure contains information about the power capabilities of the system.
POWER_STATE	The POWER_STATE union specifies a system power state value or a device power state value.
POWER_STATE	The POWER_STATE union specifies a system power state value or a device power state value.
POWER_THROTTLING_PROCESS_STATE	Stores the throttling policies and how to apply them to a target process when that process is subject to power management.
POWER_THROTTLING_THREAD_STATE	Stores the throttling policies and how to apply them to a target thread when that thread is subject to power management.
PRIVILEGE_SET	The PRIVILEGE_SET structure specifies a set of security privileges. For more information, see the reference page for PRIVILEGE_SET in the Microsoft Windows SDK documentation.
PROCESS_MITIGATION_CHILD_PROCESS_POLICY	Stores policy information about creating child processes.
PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY	Stores information about process mitigation policy.
PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY	This structure is not supported.
PROCESSOR_NUMBER	The PROCESSOR_NUMBER structure identifies a processor by its group number and group-relative processor number.
PS_CREATE_NOTIFY_INFO	The PS_CREATE_NOTIFY_INFO structure provides information about a newly created process.
REENUMERATE_SELF_INTERFACE_STANDARD	The REENUMERATE_SELF_INTERFACE_STANDARD interface structure enables a driver to request that its parent bus driver reenumerate the driver's device. This structure defines the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface.
REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION	The REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION structure contains information that a driver's RegistryCallback routine can use to free resources that the driver previously allocated for the context that is associated with a registry object.

TITLE	DESCRIPTION
REG_CREATE_KEY_INFORMATION	The REG_CREATE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key that is being created.
REG_CREATE_KEY_INFORMATION_V1	The REG_CREATE_KEY_INFORMATION_V1 structure contains information that a filter driver's RegistryCallback routine can use when a registry key is being created.
REG_DELETE_KEY_INFORMATION	The REG_DELETE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key is being deleted.
REG_DELETE_VALUE_KEY_INFORMATION	The REG_DELETE_VALUE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key's value is being deleted.
REG_ENUMERATE_KEY_INFORMATION	The REG_ENUMERATE_KEY_INFORMATION structure describes one subkey of a key whose subkeys are being enumerated.
REG_ENUMERATE_VALUE_KEY_INFORMATION	The REG_ENUMERATE_VALUE_KEY_INFORMATION structure describes one value entry of a key whose value entries are being enumerated.
REG_KEY_HANDLE_CLOSE_INFORMATION	The REG_KEY_HANDLE_CLOSE_INFORMATION structure contains information about a registry key whose handle is about to be closed.
REG_LOAD_KEY_INFORMATION	The REG_LOAD_KEY_INFORMATION structure contains information about a registry hive that is being loaded.
REG_POST_CREATE_KEY_INFORMATION	The REG_POST_CREATE_KEY_INFORMATION structure contains the result of an attempt to create a registry key.
REG_POST_OPERATION_INFORMATION	The REG_POST_OPERATION_INFORMATION structure contains information about a completed registry operation that a RegistryCallback routine can use.
REG_PRE_CREATE_KEY_INFORMATION	The REG_PRE_OPEN_KEY_INFORMATION structure contains the name of a registry key that is about to be opened.
REG_QUERY_KEY_INFORMATION	The REG_QUERY_KEY_INFORMATION structure describes the metadata that is about to be queried for a key.
REG_QUERY_KEY_NAME	The REG_QUERY_KEY_NAME structure describes the full registry key name of an object being queried.
REG_QUERY_KEY_SECURITY_INFORMATION	The REG_QUERY_KEY_SECURITY_INFORMATION structure receives security information for a registry key object.

TITLE	DESCRIPTION
REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION	The REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure describes the multiple value entries that are being retrieved for a key.
REG_QUERY_VALUE_KEY_INFORMATION	The REG_QUERY_VALUE_KEY_INFORMATION structure contains information about a registry key's value entry that is being queried.
REG_RENAME_KEY_INFORMATION	The REG_RENAME_KEY_INFORMATION structure contains the new name for a registry key whose name is about to be changed.
REG_REPLACE_KEY_INFORMATION	The REG_REPLACE_KEY_INFORMATION structure describes the metadata that is about to be replaced for a key.
REG_RESTORE_KEY_INFORMATION	The REG_RESTORE_KEY_INFORMATION structure contains the information for a registry key that is about to be restored.
REG_SAVE_KEY_INFORMATION	The REG_SAVE_KEY_INFORMATION structure contains the information for a registry key that is about to be saved.
REG_SET_INFORMATION_KEY_INFORMATION	The REG_SET_INFORMATION_KEY_INFORMATION structure describes a new setting for a key's metadata.
REG_SET_KEY_SECURITY_INFORMATION	The REG_SET_KEY_SECURITY_INFORMATION structure specifies security information for a registry key object.
REG_SET_VALUE_KEY_INFORMATION	The REG_SET_VALUE_INFORMATION structure describes a new setting for a registry key's value entry.
REG_UNLOAD_KEY_INFORMATION	The REG_UNLOAD_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry hive is unloaded.
RESOURCEMANAGER_BASIC_INFORMATION	The RESOURCEMANAGER_BASIC INFORMATION structure contains information about a resource manager object.
RESOURCEMANAGER_COMPLETION_INFORMATION	The RESOURCEMANAGER_COMPLETION_INFORMATION structure is not used.
SCATTER_GATHER_LIST	The SCATTER_GATHER_LIST structure describes the scatter/gather list for a DMA operation.
SDEV_IDENTIFIER_INTERFACE	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
SECTION_OBJECT_POINTERS	The SECTION_OBJECT_POINTERS structure, allocated by a file system or a redirector driver, is used by the memory manager and cache manager to store file-mapping and cache-related information for a file stream.
SILO_MONITOR_REGISTRATION	This structure specifies a server silo monitor that can receive notifications about server silo events.

TITLE	DESCRIPTION
SLIST_ENTRY	An SLIST_ENTRY structure describes an entry in a sequenced singly linked list.
SYSENV_VALUE	Stores the value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.
SYSENV_VARIABLE	Stores the name a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.
SYSENV_VARIABLE_INFO	Stores the information about a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_QUERY_VARIABLE_INFO request.
SYSTEM_POWER_STATE_CONTEXT	The SYSTEM_POWER_STATE_CONTEXT structure is a partially opaque system structure that contains information about the previous system power states of a computer.
SYSTEM_POWER_STATE_CONTEXT	The SYSTEM_POWER_STATE_CONTEXT structure is a partially opaque system structure that contains information about the previous system power states of a computer.
TARGET_DEVICE_CUSTOM_NOTIFICATION	The TARGET_DEVICE_CUSTOM_NOTIFICATION structure describes a custom device event.
TARGET_DEVICE_REMOVAL_NOTIFICATION	The TARGET_DEVICE_REMOVAL_NOTIFICATION structure describes a device-removal event. The PnP manager sends this structure to a driver that registered a callback routine for notification of EventCategoryTargetDeviceChange events.
TRANSACTION_BASIC_INFORMATION	The TRANSACTION_BASIC_INFORMATION structure contains information about a transaction object.
TRANSACTION_ENLISTMENT_PAIR	The TRANSACTION_ENLISTMENT_PAIR structure contains information about an enlistment that is associated with a transaction object.
TRANSACTION_ENLISTMENTS_INFORMATION	The TRANSACTION_ENLISTMENTS_INFORMATION structure contains information about the enlistments that are associated with a transaction object.
TRANSACTION_PROPERTIES_INFORMATION	The TRANSACTION_PROPERTIES_INFORMATION structure contains a transaction object's properties.
TRANSACTIONMANAGER_BASIC_INFORMATION	The TRANSACTIONMANAGER_BASIC_INFORMATION structure contains information about a transaction manager object.
TRANSACTIONMANAGER_LOG_INFORMATION	The TRANSACTIONMANAGER_LOG_INFORMATION structure contains information about a transaction manager object.
TRANSACTIONMANAGER_LOGPATH_INFORMATION	The TRANSACTIONMANAGER_LOGPATH_INFORMATION structure contains information about a transaction manager object.

TITLE	DESCRIPTION
TRANSACTIONMANAGER_RECOVERY_INFORMATION	The TRANSACTIONMANAGER_RECOVERY_INFORMATION structure contains information about a transaction manager object.
UNICODE_STRING	The UNICODE_STRING structure is used to define Unicode strings.
WMIGUIDREGINFO	The WMIGUIDREGINFO structure contains registration information for a given data block or event block exposed by a driver that uses the WMI library support routines.
WMILIB_CONTEXT	The WMILIB_CONTEXT structure provides registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.
WMIREGGUIDW	The WMIREGGUID structure contains new or updated registration information for a data block or event block.
WMIREGINFOW	The WMIREGINFO structure contains information provided by a driver to register or update its data blocks and event blocks.
WNODE_ALL_DATA	The WNODE_ALL_DATA structure contains data for all instances of a data block or event block.
WNODE_EVENT_ITEM	The WNODE_EVENT_ITEM structure contains data generated by a driver for an event.
WNODE_EVENT_REFERENCE	The WNODE_EVENT_REFERENCE structure contains information that WMI can use to query for an event that exceeds the event size limit set in the registry.
WNODE_HEADER	The WNODE_HEADER structure is the first member of all other WNODE_XXX structures. It contains information common to all such structures.
WNODE_METHOD_ITEM	The WNODE_METHOD_ITEM structure indicates a method associated with an instance of a data block and contains any input data for the method.
WNODE_SINGLE_INSTANCE	The WNODE_SINGLE_INSTANCE structure contains values for all data items in one instance of a data block.
WNODE_SINGLE_ITEM	The WNODE_SINGLE_ITEM structure contains the value of a single data item in an instance of a data block.
WNODE_TOO_SMALL	The WNODE_TOO_SMALL structure indicates the size of the buffer needed to receive output from a request.
XVARIABLE_NAME	Stores the name of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES request.

TITLE	DESCRIPTION
XVARIABLE_NAME_AND_VALUE	Stores the name and value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES and IOCTL_SYSENV_SET_VARIABLE requests.

aux_klib.h header

1/8/2019 • 2 minutes to read • Edit Online

This header is used by kernel. For more information, see:

- Windows kernel aux_klib.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
AuxKlibEnumerateSystemFirmwareTables	The AuxKlibEnumerateSystemFirmwareTables routine enumerates all system firmware tables of the specified type.
AuxKlibGetBugCheckData	The AuxKlibGetBugCheckData routine retrieves information about a bug check that has just occurred.
AuxKlibGetImageExportDirectory	The AuxKlibGetImageExportDirectory routine returns an image module's export directory.
AuxKlibGetSystemFirmwareTable	The AuxKlibGetSystemFirmwareTable routine retrieves the specified firmware table from the firmware table provider.
AuxKlibInitialize	The AuxKlibInitialize routine initializes the Auxiliary Kernel-Mode Library.
AuxKlibQueryModuleInformation	The AuxKlibQueryModuleInformation routine retrieves information about the image modules that the operating system has loaded.

Structures

TITLE	DESCRIPTION
AUX_MODULE_BASIC_INFO	The AUX_MODULE_BASIC_INFO structure contains basic information about a loaded image module.
AUX_MODULE_EXTENDED_INFO	The AUX_MODULE_EXTENDED_INFO structure contains extended information about a loaded image module.
KBUGCHECK_DATA	The KBUGCHECK_DATA structure contains bug check parameters.

AUX_MODULE_BASIC_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **AUX_MODULE_BASIC_INFO** structure contains basic information about a loaded image module.

Syntax

```
typedef struct _AUX_MODULE_BASIC_INFO {
  PVOID ImageBase;
} AUX_MODULE_BASIC_INFO, *PAUX_MODULE_BASIC_INFO;
```

Members

`ImageBase`

A pointer to the base of the loaded image.

Remarks

The **AUX_MODULE_BASIC_INFO** structure is used as input to the AuxKlibQueryModuleInformation routine.

Requirements

Header	aux_klib.h (include Aux_klib.h)

See Also

AuxKlibQueryModuleInformation

AUX_MODULE_EXTENDED_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **AUX_MODULE_EXTENDED_INFO** structure contains extended information about a loaded image module.

Syntax

```
typedef struct _AUX_MODULE_EXTENDED_INFO {
  AUX_MODULE_BASIC_INFO BasicInfo;
  ULONG                 ImageSize;
  USHORT                FileNameOffset;
  UCHAR                 FullPathName[AUX_KLIB_MODULE_PATH_LEN];
} AUX_MODULE_EXTENDED_INFO, *PAUX_MODULE_EXTENDED_INFO;
```

Members

`BasicInfo`

An AUX_MODULE_BASIC_INFO structure.

`ImageSize`

The size, in bytes, of the loaded image.

`FileNameOffset`

The offset, in bytes, from the beginning of the full path name to the file name of the module's image file.

`FullPathName`

The full path name of the module's image file.

Remarks

The **AUX_MODULE_EXTENDED_INFO** structure is used as input to the AuxKlibQueryModuleInformation routine.

Requirements

Header	aux_klib.h (include Aux_klib.h)

See Also

AuxKlibQueryModuleInformation

KBUGCHECK_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_DATA** structure contains bug check parameters.

Syntax

```
typedef struct _KBUGCHECK_DATA {
  ULONG     BugCheckDataSize;
  ULONG     BugCheckCode;
  ULONG_PTR Parameter1;
  ULONG_PTR Parameter2;
  ULONG_PTR Parameter3;
  ULONG_PTR Parameter4;
} KBUGCHECK_DATA, *PKBUGCHECK_DATA;
```

Members

`BugCheckDataSize`

The size, in bytes, of the **KBUGCHECK_DATA** structure. Callers of AuxKlibGetBugCheckData must supply this value.

`BugCheckCode`

The bug check code. This value identifies that type of bug check that has occurred.

`Parameter1`

Bug check parameter 1. For more information about this parameter, see specific bug check codes.

`Parameter2`

Bug check parameter 2. For more information about this parameter, see specific bug check codes.

`Parameter3`

Bug check parameter 3. For more information about this parameter, see specific bug check codes.

`Parameter4`

Bug check parameter 4. For more information about this parameter, see specific bug check codes.

Remarks

The **KBUGCHECK_DATA** structure is used as a parameter to AuxKlibGetBugCheckData.

Requirements

Header	aux_klib.h (include Aux_klib.h)

See Also

AuxKlibGetBugCheckData

AuxKlibEnumerateSystemFirmwareTables function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibEnumerateSystemFirmwareTables** routine enumerates all system firmware tables of the specified type.

Syntax

```
NTSTATUS AuxKlibEnumerateSystemFirmwareTables(
  ULONG  FirmwareTableProviderSignature,
  PVOID  FirmwareTableBuffer,
  ULONG  BufferLength,
  PULONG ReturnLength
);
```

Parameters

`FirmwareTableProviderSignature`

The identifier of the firmware table provider to which the query is to be directed. This parameter can be one of the following values.

VALUE	MEANING
'ACPI'	The ACPI firmware table provider.
'FIRM'	The raw firmware table provider.
'RSMB'	The raw SMBIOS firmware table provider.

`FirmwareTableBuffer`

A pointer to a caller-allocated buffer to which the routine writes the list of firmware tables. If this parameter is NULL, the value written to **ReturnLength* is the required buffer size. For more information about the contents of this buffer, see the Remarks section.

`BufferLength`

The size, in bytes, of the buffer pointed to by *FirmwareTableBuffer*.

`ReturnLength`

A pointer to a location to which the routine writes the number of bytes of data written to the buffer pointed to by *FirmwareTableBuffer*.

Return Value

AuxKlibEnumerateSystemFirmwareTables returns STATUS_SUCCESS if the call is successful. Possible return values include the following error codes.

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	An invalid parameter was supplied to the routine.
STATUS_BUFFER_TOO_SMALL	The caller-allocated buffer is too small, but the required buffer size has been written to the *ReturnLength* output parameter.

Remarks

If the call is successful, this routine writes an array of one or more table identifiers to the buffer pointed to by the *FirmwareTableBuffer* parameter. Each element of this array is a DWORD value that identifies a firmware table that is available from the provider specified by the *FirmwareTableProviderSignature* parameter.

The raw SMBIOS table provider ('RSMB') currently returns a single table identifier, 0x00000000. This corresponds to the raw SMBIOS firmware table.

The raw firmware table provider ('FIRM') retrieves a list of DWORD table identifiers. Each identifier corresponds to the beginning of a physical address range. Currently, this provider returns two identifiers, 0x000C0000 and 0x000E0000. These identifiers represent physical memory from 0x000C0000 to 0x000DFFFF and from 0x000E0000 to 0x000FFFFF, respectively.

The ACPI table provider ('ACPI') returns a list of DWORD table identifiers. Each identifier returned corresponds to **Signature** field of the **DESCRIPTION_HEADER** structure for an ACPI table currently in the ACPI namespace of the system. For more information about this structure, see the Advanced Configuration and Power Interface Specification at the Advanced Configuration and Power Interface website.

For ACPI, if the system firmware contains multiple tables with the same name, **AuxKlibEnumerateSystemFirmwareTables** enumerates them all. However, AuxKlibGetSystemFirmwareTable retrieves only the first table in the list that has this name.

AuxKlibEnumerateSystemFirmwareTables is the kernel-mode equivalent of the Win32 EnumSystemFirmwareTables function.

Drivers must call AuxKlibInitialize before calling **AuxKlibEnumerateSystemFirmwareTable**.

AuxKlibEnumerateSystemFirmwareTables is defined in the Aux_klib.h header file that is included in the WDK for Windows 8 and later versions of Windows. To use this routine, drivers should link to the version of Aux_klib.lib that is included in the WDK for Windows 8 and later versions of Windows. The implementation of **AuxKlibEnumerateSystemFirmwareTables** in these versions of the WDK can be used in versions of Windows starting with Windows Vista.

Requirements

Minimum supported client	Supported starting with Windows Vista.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)
Library	Aux_Klib.lib
IRQL	PASSIVE_LEVEL

See Also

AuxKlibGetSystemFirmwareTable

AuxKlibInitialize

EnumSystemFirmwareTables

AuxKlibGetBugCheckData function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibGetBugCheckData** routine retrieves information about a bug check that has just occurred.

Syntax

```
NTSTATUS AuxKlibGetBugCheckData(
  PKBUGCHECK_DATA BugCheckData
);
```

Parameters

`BugCheckData`

A pointer to a KBUGCHECK_DATA structure that contains information about the bug check. The *BugCheckData* size of this structure should be set equal to the size, in bytes, of the **KBUGCHECK_DATA** structure.

Return Value

AuxKlibGetBugCheckData returns STATUS_SUCCESS if the operation succeeds. The routine returns STATUS_INFO_LENGTH_MISMATCH if the **KBUGCHECK_DATA** structure's size is incorrect.

Remarks

The **AuxKlibGetBugCheckData** routine can be called only from a BugCheckCallback routine.

Drivers must call AuxKlibInitialize before calling **AuxKlibGetBugCheckData**.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)
Library	Aux_Klib.lib

See Also

AuxKlibInitialize

BugCheckCallback

KBUGCHECK_DATA

AuxKlibGetImageExportDirectory function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibGetImageExportDirectory** routine returns an image module's export directory.

Syntax

```
PIMAGE_EXPORT_DIRECTORY AuxKlibGetImageExportDirectory(
  PVOID ImageBase
);
```

Parameters

`ImageBase`

A pointer to the base of an image, which is obtained by calling AuxKlibQueryModuleInformation.

Return Value

AuxKlibGetImageExportDirectory returns a pointer to an **IMAGE_EXPORT_DIRECTORY** structure. This structure is defined in Ntimage.h, which is included in the Microsoft Windows Driver Kit (WDK).

Remarks

Drivers must call AuxKlibInitialize before calling **AuxKlibGetImageExportDirectory**.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)
Library	Aux_Klib.lib

See Also

AuxKlibQueryModuleInformation

AuxKlibGetSystemFirmwareTable function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibGetSystemFirmwareTable** routine retrieves the specified firmware table from the firmware table provider.

Syntax

```
NTSTATUS AuxKlibGetSystemFirmwareTable(
  ULONG  FirmwareTableProviderSignature,
  ULONG  FirmwareTableID,
  PVOID  FirmwareTableBuffer,
  ULONG  BufferLength,
  PULONG ReturnLength
);
```

Parameters

`FirmwareTableProviderSignature`

The identifier of the firmware table provider to which the query is to be directed. This parameter can be one of the following values.

VALUE	MEANING
'ACPI'	The ACPI firmware table provider.
'FIRM'	The raw firmware table provider.
'RSMB'	The raw SMBIOS firmware table provider.

`FirmwareTableID`

The identifier of the firmware table. The characters in the identifier are in little-endian order.

For example, FACP is the name of a table provided by ACPI. The FACP table is identified by the 4-byte value 'PCAF' (0x50434146) in the **Signature** field of the **DESCRIPTION_HEADER** structure at the start of the table. The following code example shows how to specify FACP in an **AuxKlibGetSystemFirmwareTable** call:

```
status = AuxKlibGetSystemFirmwareTable('ACPI', 'PCAF', pBuffer, BUFSIZE, &dataSize);
```

In this example, `pBuffer` points to the caller-allocated buffer, `BUFSIZE` is the size in bytes of this buffer, and `dataSize` is a variable to which the routine writes the number of bytes written to the buffer. For more information about the **DESCRIPTION_HEADER** structure, see the Advanced Configuration and Power Interface Specification at the Advanced Configuration and Power Interface website.

`FirmwareTableBuffer`

A pointer to a caller-allocated buffer that receives the list of firmware tables. If this parameter is NULL, the value written to **ReturnLength* is the required buffer size. For more information about the contents of this buffer, see the Remarks section.

`BufferLength`

The size, in bytes, of the buffer pointed to by *FirmwareTableBuffer*.

`ReturnLength`

A pointer to a location to which the routine writes the number of bytes of data written to the buffer pointed to by *FirmwareTableBuffer*.

Return Value

AuxKlibGetSystemFirmwareTable returns STATUS_SUCCESS if the call is successful. Possible return values include the following error codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	An invalid parameter was supplied to the routine.
STATUS_BUFFER_TOO_SMALL	The caller-allocated buffer is too small, but the required buffer size has been written to the *ReturnLength* output parameter.

Remarks

The raw SMBIOS table provider ('RSMB') retrieves the contents of the raw SMBIOS firmware table. The data written to the *FirmwareTableBuffer* buffer starts with the following structure:

```
struct RawSMBIOSData
{
  BYTE  Used20CallingMethod;
  BYTE  SMBIOSMajorVersion;
  BYTE  SMBIOSMinorVersion;
  BYTE  DmiRevision;
  DWORD  Length;
  BYTE  SMBIOSTableData[];
};
```

The raw firmware table provider ('FIRM') retrieves the contents of the specified physical address range. The value written to **ReturnLength* is the size of the address range.

The ACPI table provider ('ACPI') retrieves the contents of the specified ACPI table. Because OEMs can include ACPI firmware tables that are not listed in the ACPI specification, you should first call

AuxKlibEnumerateSystemFirmwareTables to enumerate all ACPI tables that are currently available from the system firmware.

For ACPI, if the system firmware contains multiple tables with the same name, **AuxKlibEnumerateSystemFirmwareTables** enumerates them all. However, **AuxKlibGetSystemFirmwareTable** retrieves only the first table in the list that has this name.

AuxKlibGetSystemFirmwareTable is the kernel-mode equivalent of the Win32 GetSystemFirmwareTable function.

Drivers must call AuxKlibInitialize before calling **AuxKlibGetSystemFirmwareTable**.

AuxKlibGetSystemFirmwareTable is defined in the Aux_klib.h header file that is included in the WDK for Windows 8 and later versions of Windows. To use this routine, drivers should link to the version of Aux_klib.lib that is included in the WDK for Windows 8 and later versions of Windows. The implementation of **AuxKlibGetSystemFirmwareTable** in these versions of the WDK can be used in versions of Windows starting with Windows Vista.

Requirements

Minimum supported client	Supported starting with Windows Vista.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)
Library	Aux_Klib.lib
IRQL	PASSIVE_LEVEL

See Also

AuxKlibEnumerateSystemFirmwareTables

AuxKlibInitialize

GetSystemFirmwareTable

AuxKlibInitialize function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibInitialize** routine initializes the Auxiliary Kernel-Mode Library.

Syntax

```
NTSTATUS AuxKlibInitialize(

);
```

Parameters

This function has no parameters.

Return Value

AuxKlibInitialize returns STATUS_SUCCESS if the operation succeeds. Otherwise, the routine returns an appropriate NTSTATUS value.

Remarks

Drivers that use the Auxiliary Kernel-Mode Library must call **AuxKlibInitialize** before calling any of the library's other routines.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)
Library	Aux_Klib.lib

AuxKlibQueryModuleInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **AuxKlibQueryModuleInformation** routine retrieves information about the image modules that the operating system has loaded.

Syntax

```
NTSTATUS AuxKlibQueryModuleInformation(
  PULONG BufferSize,
  ULONG  ElementSize,
  PVOID  QueryInfo
);
```

Parameters

`BufferSize`

A pointer to a location that contains or receives a buffer size, in bytes. If *QueryInfo* is **NULL**, the location receives the number of bytes that the driver must allocate for the array that receives the retrieved information. If *QueryInfo* is not **NULL**, the location must contain the specified number of bytes.

`ElementSize`

The size, in bytes, of each element of the array that *QueryInfo* points to. This value must be **sizeof**(**AUX_MODULE_BASIC_INFO**) or **sizeof**(**AUX_MODULE_EXTENDED_INFO**).

`QueryInfo`

A pointer to an array of AUX_MODULE_BASIC_INFO or AUX_MODULE_EXTENDED_INFO structures that receives information about loaded image modules. If this pointer is **NULL**, **AuxKlibQueryModuleInformation** writes the required buffer size to the location that *BufferSize* points to.

Return Value

AuxKlibQueryModuleInformation returns STATUS_SUCCESS if the operation succeeds. **AuxKlibQueryModuleInformation** returns STATUS_BUFFER_TOO_SMALL if the *QueryInfo* pointer is not **NULL** and the driver-supplied *BufferSize* value is too small.

The routine might return other NTSTATUS values.

Remarks

To obtain information about the operating system's loaded image modules, a driver must:

1. Call **AuxKlibQueryModuleInformation** with a **NULL** *QueryInfo* pointer. After **AuxKlibQueryModuleInformation** returns, the location that the *BufferSize* parameter points to will contain the number of bytes that the driver will have to allocate for the array.
2. Call a memory allocation routine, such as ExAllocatePoolWithTag, to allocate a buffer for the array.
3. Call **AuxKlibQueryModuleInformation** again. This time, the *QueryInfo* pointer must contain the address of the allocated buffer. After **AuxKlibQueryModuleInformation** returns, the buffer contains an array of module

information.

The number of loaded modules can change between the first and second calls to **AuxKlibQueryModuleInformation**. As a result, the second call to **AuxKlibQueryModuleInformation** might return STATUS_BUFFER_TOO_SMALL even if the driver allocates a buffer that is based on the size that was obtained from the first call.

If a call to **AuxKlibQueryModuleInformation** succeeds, the routine writes an **ImageBase** value to each element in the *QueryInfo* array. Each **ImageBase** value is a pointer to the base of a loaded driver image. This pointer remains valid only while the driver remains loaded. The caller should assume that the driver can be unloaded at any time unless the caller can guarantee otherwise. For example, a driver might be unloaded between a call to **AuxKlibQueryModuleInformation** that obtains a pointer to the driver image and a call to AuxKlibGetImageExportDirectory that uses this pointer.

Drivers must call AuxKlibInitialize before calling **AuxKlibQueryModuleInformation**.

Examples

The following code example illustrates the steps that are listed in the preceding Remarks section.

```
NTSTATUS  status;
ULONG  modulesSize;
AUX_MODULE_EXTENDED_INFO*  modules;
ULONG  numberOfModules;
//
// Get the required array size.
//
status = AuxKlibQueryModuleInformation(
                                       &modulesSize,
                                       sizeof(AUX_MODULE_EXTENDED_INFO),
                                       NULL
                                       );

if (!NT_SUCCESS(status) || modulesSize == 0) {
    break;
    }

//
// Calculate the number of modules.
//
numberOfModules = modulesSize / sizeof(AUX_MODULE_EXTENDED_INFO);

//
// Allocate memory to receive data.
//
modules =
    (AUX_MODULE_EXTENDED_INFO*) ExAllocatePoolWithTag(
                                     PagedPool,
                                     modulesSize,
                                     '3LxF'
                                     );
if (modules == NULL) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
    }

RtlZeroMemory(
              modules,
              modulesSize
              );

//
// Obtain the module information.
//
status = AuxKlibQueryModuleInformation(
                                       &modulesSize,
                                       sizeof(AUX_MODULE_EXTENDED_INFO),
                                       modules
                                       );
if (!NT_SUCCESS(status)) {
    break;
    }
```

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Universal
Header	aux_klib.h (include Aux_klib.h)

Library	Aux_Klib.lib

See Also

AUX_MODULE_BASIC_INFO

AUX_MODULE_EXTENDED_INFO

AuxKlibGetImageExportDirectory

AuxKlibInitialize

ExAllocatePoolWithTag

buffring.h header

1/8/2019 • 2 minutes to read • Edit Online

This header defines the change buffering state requests related data structures in RDBSS. For information about the programming interfaces, see the header comments.

- Windows kernel

ioaccess.h header

1/8/2019 • 2 minutes to read • Edit Online

Thsi header defines function prototypes for accessing I/O ports and memory on I/O adapters.

For information about accessing device registers, see:

- Reading and Writing to Device Registers ioaccess.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.

READ_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_UCHAR** routine reads a byte from the specified port address.

Syntax

```
UCHAR READ_PORT_UCHAR(
  PVOID Port
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

Return Value

READ_PORT_UCHAR returns the byte that is read from the specified port address.

Remarks

Callers of **READ_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_ULONG** routine reads a ULONG value from the specified port address.

Syntax

```
ULONG READ_PORT_ULONG(
  PVOID Port
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped range in I/O space.

Return Value

READ_PORT_ULONG returns the ULONG value that is read from the specified port address.

Remarks

Callers of **READ_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_USHORT** routine reads a USHORT value from the specified port address.

Syntax

```
USHORT READ_PORT_USHORT(
  PVOID Port
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped range in I/O space.

Return Value

READ_PORT_USHORT returns the USHORT value that is read from the specified port address.

Remarks

Callers of **READ_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_UCHAR** routine reads a byte from the specified register address.

Syntax

```
UCHAR READ_REGISTER_UCHAR(
  _In_ PUCHAR Register
);
```

Parameters

`Register`

Pointer to the register address, which must be a mapped range in memory space.

Return Value

READ_REGISTER_UCHAR returns the byte read from the specified register address.

Remarks

Callers of **READ_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_ULONG** routine reads a ULONG value from the specified register address.

Syntax

```
ULONG READ_REGISTER_ULONG(
  _In_ PULONG Register
);
```

Parameters

`Register`

Pointer to the register address, which must be a mapped range in memory space.

Return Value

READ_REGISTER_ULONG returns the ULONG value read from the specified register address.

Remarks

Callers of **READ_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_USHORT** routine reads a USHORT value from the specified register address.

Syntax

```
USHORT READ_REGISTER_USHORT(
  _In_ PUSHORT Register
);
```

Parameters

`Register`

Pointer to the register address, which must be a mapped range in memory space.

Return Value

READ_REGISTER_USHORT returns the USHORT value read from the specified register address.

Remarks

Callers of **READ_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_UCHAR** routine writes a byte to the specified port address.

Syntax

```
void WRITE_PORT_UCHAR(
  PVOID Port,
  ULONG Value
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Value`

Specifies a byte to be written to the port.

Return Value

None

Remarks

Callers of **WRITE_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_ULONG** routine writes a ULONG value to the specified port address.

Syntax

```
void WRITE_PORT_ULONG(
  PVOID Port,
  ULONG Value
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Value`

Specifies a ULONG value to be written to the port.

Return Value

None

Remarks

Callers of **WRITE_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_USHORT** routine writes a USHORT value to the specified port address.

Syntax

```
void WRITE_PORT_USHORT(
  PVOID Port,
  ULONG Value
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Value`

Specifies a USHORT value to be written to the port.

Return Value

None

Remarks

Callers of **WRITE_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_UCHAR** routine writes a byte to the specified address.

Syntax

```
VOID WRITE_REGISTER_UCHAR(
  _In_ PUCHAR Register,
  _In_ UCHAR  Value
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Value`

Specifies a byte to be written to the register.

Return Value

None

Remarks

Callers of **WRITE_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_ULONG** routine writes a ULONG value to the specified address.

Syntax

```
VOID WRITE_REGISTER_ULONG(
  _In_ PULONG Register,
  _In_ ULONG  Value
);
```

Parameters

`Register`

Pointer to the register which must be a mapped range in memory space.

`Value`

Specifies a ULONG value to be written to the register.

Return Value

None

Remarks

Callers of **WRITE_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_USHORT** routine writes a USHORT value to the specified address.

Syntax

```
VOID WRITE_REGISTER_USHORT(
  _In_ PUSHORT Register,
  _In_ USHORT  Value
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Value`

Specifies a USHORT value to be written to the register.

Return Value

None

Remarks

Callers of **WRITE_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ioaccess.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

iointex.h header

1/8/2019 • 2 minutes to read • Edit Online

This header exposes the programming interfaces to connect and disconnect interrupts allowing drivers to use those interfaces to run on downlevel systems. For more information, see:

- Registering an ISR
- Servicing Interrupts iointex.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
WdmlibIoConnectInterruptEx	The WdmlibIoConnectInterruptEx function registers an interrupt-handling routine for a device's interrupts.
WdmlibIoDisconnectInterruptEx	The WdmlibIoDisconnectInterruptEx function unregisters an interrupt service routine (ISR) that was registered by a previous call to the WdmlibIoConnectInterruptEx function.
WdmlibIoGetAffinityInterrupt	The WdmlibIoGetAffinityInterrupt function gets the group affinity of an interrupt object.

WdmlibIoConnectInterruptEx function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibIoConnectInterruptEx** function registers an interrupt-handling routine for a device's interrupts.

Syntax

```
NTSTATUS WdmlibIoConnectInterruptEx(
  PIO_CONNECT_INTERRUPT_PARAMETERS Parameters
);
```

Parameters

`Parameters`

Pointer to an IO_CONNECT_INTERRUPT_PARAMETERS structure that specifies the device and interrupt-handling routine. On return, **WdmlibIoConnectInterruptEx** updates this structure to hold information about the device's interrupts.

Return Value

The function returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error value on failure. Possible error values include:

RETURN CODE	DESCRIPTION
STATUS_INVALID_DEVICE_REQUEST	The operation is invalid for the specified device. For example, *Parameters*->**Version** = CONNECT_LINE_BASED, and the system has assigned multiple interrupt messages to the device.
STATUS_INVALID_PARAMETER	The caller specified an invalid parameter. This error occurs, for example, when the caller specified **NULL** for the device's device object.

STATUS_INVALID_PARAMETER_1	The caller specified an invalid value for the **Version** member of the structure that *Parameters* points to. *Parameters*->**Version** must be one of CONNECT_LINE_BASED, CONNECT_MESSAGE_BASED, or CONNECT_FULLY_SPECIFIED.
STATUS_INVALID_PARAMETER_10	The caller specified an invalid value for the tenth member of the structure that *Parameters* points to. This error occurs, for example, when *Parameters*->**FullySpecified.ProcessorEnableMask** does not have any bits set.
STATUS_NOT_FOUND	One of the arguments was not found. For example, the specified device has no interrupts, or the specified interrupt vector is not assigned to any device.

Remarks

WdmlibIoConnectInterruptEx can be used to register an interrupt-handling routine for both traditional line-based interrupts (such as that supported by the PCI bus), and the newer message-signaled interrupts (such as that supported by PCI versions 2.2 and 3.0).

Drivers register an InterruptService routine for line-based interrupts, and an InterruptMessageService routine for message-signaled interrupts. For more information about how to specify the members of *Parameters* in each case, see IO_CONNECT_INTERRUPT_PARAMETERS.

WdmlibIoConnectInterruptEx updates the members of *Parameters* to provide information about the device's interrupts. For more information about the information provided by **WdmlibIoConnectInterruptEx**, see IO_CONNECT_INTERRUPT_PARAMETERS.

Use WdmlibIoDisconnectInterruptEx to unregister a routine registered with **WdmlibIoConnectInterruptEx**.

The driver should not program its device to generate interrupts until it has connected its ISR. Thus, the ISR cannot fire before **WdmlibIoConnectInterruptEx** returns. However, there are certain devices, such as buttons, that are not programmable. For those devices, the driver should be prepared to handle the ISR as soon as it calls **WdmlibIoConnectInterruptEx**. The interrupt line may already be asserted when **WdmlibIoConnectInterruptEx** is called and can fire immediately after the line is enabled at the interrupt controller, before the **WdmlibIoConnectInterruptEx** call unwinds.

For more information about registering an interrupt-handling routine, see Registering an ISR.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system. Drivers that must also work on Windows 2000, Windows XP, or Windows Server 2003 can instead link to Iointex.lib to use the routine.
Target Platform	Universal
Header	iointex.h (include Iointex.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IO_CONNECT_INTERRUPT_PARAMETERS

InterruptMessageService

InterruptService

WdmlibIoDisconnectInterruptEx

WdmlibIoDisconnectInterruptEx function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibIoDisconnectInterruptEx** function unregisters an interrupt service routine (ISR) that was registered by a previous call to the WdmlibIoConnectInterruptEx function.

Syntax

```
NTSTATUS WdmlibIoDisconnectInterruptEx(
  PIO_DISCONNECT_INTERRUPT_PARAMETERS Parameters
);
```

Parameters

`Parameters`

Pointer to an IO_DISCONNECT_INTERRUPT_PARAMETERS structure that contains the connection context for the interrupt being disconnected.

Return Value

None.

Remarks

The driver should configure the device to issue interrupts only when these interrupts are connected. Failure to prevent a device from issuing interrupts when the interrupts are disconnected might cause system instability. For example, if a device shares a level-triggered interrupt line with other devices, and the device issues an interrupt request when the device's interrupts are disconnected, the other devices on the line will not acknowledge the interrupt and the interrupt will continue to fire. Before calling **WdmlibIoDisconnectInterruptEx**, the driver should configure the device to stop issuing interrupts. After calling WdmlibIoConnectInterruptEx, the driver should configure the device to start issuing interrupts.

Requirements

Minimum supported client	Available starting with Windows Vista. Drivers that must also work Microsoft Windows 2000, Windows XP, or Windows Server 2003 can instead link to Iointex.lib to use the routine.
Target Platform	Universal
Header	iointex.h (include Iointex.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib; Iointex.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL

See Also

IO_DISCONNECT_INTERRUPT_PARAMETERS

WdmlibIoConnectInterruptEx

WdmlibIoGetAffinityInterrupt function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibIoGetAffinityInterrupt** function gets the group affinity of an interrupt object.

Syntax

```
NTSTATUS WdmlibIoGetAffinityInterrupt(
  PKINTERRUPT     InterruptObject,
  PGROUP_AFFINITY GroupAffinity
);
```

Parameters

`InterruptObject`

A pointer to an interrupt object. This parameter points to a KINTERRUPT structure that represents a registration by a driver to receive device interrupts. The structure is opaque. The caller obtained this pointer value in a previous call to the WdmlibIoConnectInterruptEx or IoConnectInterrupt routine.

`GroupAffinity`

A pointer to a caller-allocated buffer into which the routine writes a GROUP_AFFINITY structure that specifies the group affinity of the interrupt object that *InterruptObject* points to. This buffer must be large enough to contain the structure.

Return Value

WdmlibIoGetAffinityInterrupt returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *InterruptObject* parameter does not point to a valid interrupt object.

Remarks

A kernel-mode driver calls this routine to get the set of logical processors on which the driver's registered interrupt service routine (ISR) can receive device interrupts. This set of processors is described by a **GROUP_AFFINITY** structure, which specifies a group number and an affinity mask. All the processors that are assigned to a particular ISR registration must belong to the same group.

The driver registered the ISR in a previous call to the WdmlibIoConnectInterruptEx or **IoConnectInterrupt**

routine.

In Windows 7, WdmlibIoConnectInterruptEx and **IoConnectInterrupt** assign device interrupts only to logical processors in group 0. This is by default. A driver can specify a different interrupt affinity for its device in an INF file or in its response to an IRP_MN_FILTER_RESOURCE_REQUIREMENTS request. For more information about how to change the interrupt affinity, see the Supporting Systems That Have More Than 64 Processors white paper on the WHDC website.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Windows
Header	iointex.h (include Iointex.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

GROUP_AFFINITY

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

IoConnectInterrupt

KINTERRUPT

WdmlibIoConnectInterruptEx

miniport.h header

1/8/2019 • 9 minutes to read • Edit Online

This header is used by storage. For more information, see:

- Storage miniport.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
ARM64_SYSREG_CRM	Returns the CRm field for a given ARM system register encoding value. For more information, see the ARM Architecture Reference Manual.
ARM64_SYSREG_CRN	Returns the CRn field for a given ARM system register encoding value. For more information, see the ARM Architecture Reference Manual.
ARM64_SYSREG_OP1	Returns the op1 field for a given ARM system register encoding value. For more information, see the ARM Architecture Reference Manual.
ARM64_SYSREG_OP2	Returns the op2 field for a given ARM system register encoding value. For more information, see the ARM Architecture Reference Manual.
FIELD_OFFSET	The FIELD_OFFSET macro returns the byte offset of a named field in a known structure type.
InterlockedAnd	The InterlockedAnd macro atomically computes a bitwise AND operation.
InterlockedCompareExchange	The InterlockedCompareExchange routine performs an atomic operation that compares the input value pointed to by Destination with the value of Comparand.
InterlockedCompareExchangePointer	The InterlockedCompareExchangePointer routine performs an atomic operation that compares the input pointer value pointed to by Destination with the pointer value Comparand.
InterlockedDecrement	The InterlockedDecrement routine decrements a caller-supplied variable of type LONG as an atomic operation.
InterlockedExchange	The InterlockedExchange routine sets an integer variable to a given value as an atomic operation.
InterlockedExchangeAdd	The InterlockedExchangeAdd routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.

TITLE	DESCRIPTION
InterlockedExchangePointer	The InterlockedExchangePointer routine performs an atomic operation that sets a pointer to a new value.
InterlockedIncrement	The InterlockedIncrement routine increments a caller-supplied variable as an atomic operation.
InterlockedOr	The InterlockedOr routine atomically computes a bitwise OR operation.
InterlockedXor	The InterlockedOr routine atomically computes a bitwise exclusive OR operation.
READ_PORT_BUFFER_UCHAR	The READ_PORT_BUFFER_UCHAR routine reads a number of bytes from the specified port address into a buffer.
READ_PORT_BUFFER_ULONG	The READ_PORT_BUFFER_ULONG routine reads a number of ULONG values from the specified port address into a buffer.
READ_PORT_BUFFER_USHORT	The READ_PORT_BUFFER_USHORT routine reads a number of USHORT values from the specified port address into a buffer.
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG64	The READ_REGISTER_BUFFER_ULONG64 function reads a number of ULONG64 values from the specified register address into a buffer.
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG64	The READ_REGISTER_ULONG64 function reads a ULONG64 value from the specified register address.

TITLE	DESCRIPTION
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
WRITE_PORT_BUFFER_UCHAR	The WRITE_PORT_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified port.
WRITE_PORT_BUFFER_ULONG	The WRITE_PORT_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified port address.
WRITE_PORT_BUFFER_USHORT	The WRITE_PORT_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified port address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG64	The WRITE_REGISTER_BUFFER_ULONG64 function writes a number of ULONG64 values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_ULONG64	The WRITE_REGISTER_ULONG64 function writes a ULONG64 value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.

Callback functions

TITLE	DESCRIPTION

TITLE	DESCRIPTION
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.

Structures

TITLE	DESCRIPTION
CM_EISA_FUNCTION_INFORMATION	The CM_EISA_FUNCTION_INFORMATION structure defines detailed EISA configuration information returned by HalGetBusData for the input BusDataType EisaConfiguration, or by HalGetBusDataByOffset for the input BusDataType EisaConfiguration and the Offset zero, assuming the caller-allocated Buffer is of sufficient Length.
CM_EISA_SLOT_INFORMATION	The CM_EISA_SLOT_INFORMATION structure defines EISA configuration header information returned by HalGetBusData for the input BusDataType = EisaConfiguration, or by HalGetBusDataByOffset for the inputs BusDataType = EisaConfiguration and Offset = 0, assuming the caller-allocated Buffer is of sufficient Length.
CM_MCA_POS_DATA	The CM_MCA_POS_DATA structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.
EMULATOR_ACCESS_ENTRY	The EMULATOR_ACCESS_ENTRY structure specifies an element in the EmulatorAccessEntries array set up in the VIDEO_PORT_CONFIG_INFO structure by drivers of VGA-compatible (SVGA) adapters on x86-based NT-based operating system platforms.
GROUP_AFFINITY	The GROUP_AFFINITY structure specifies a group number and the processor affinity within that group.
INTERFACE	The INTERFACE structure describes an interface that is exported by a driver for use by other drivers.
IO_RESOURCE_DESCRIPTOR	The IO_RESOURCE_DESCRIPTOR structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of IO_RESOURCE_DESCRIPTOR structures is contained within each IO_RESOURCE_LIST structure.
IO_RESOURCE_LIST	The IO_RESOURCE_LIST structure describes a range of raw hardware resources, of various types, that can be used by a device.
IO_RESOURCE_REQUIREMENTS_LIST	The IO_RESOURCE_REQUIREMENTS_LIST structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.
PCI_CAPABILITIES_HEADER	The PCI_CAPABILITIES_HEADER structure defines a header that is present in every PCI capability structure.

TITLE	DESCRIPTION
PCI_COMMON_CONFIG	The PCI_COMMON_CONFIG structure is obsolete.
PCI_EXPRESS_AER_CAPABILITIES	The PCI_EXPRESS_AER_CAPABILITIES structure describes a PCI Express (PCIe) advanced error capabilities and control register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_AER_CAPABILITY	The PCI_EXPRESS_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure.
PCI_EXPRESS_BRIDGE_AER_CAPABILITY	The PCI_EXPRESS_BRIDGE_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure for a PCIe bridge device.
PCI_EXPRESS_CAPABILITIES_REGISTER	The PCI_EXPRESS_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) capabilities register of a PCIe capability structure.
PCI_EXPRESS_CAPABILITY	The PCI_EXPRESS_CAPABILITY structure describes a PCI Express (PCIe) capability structure.
PCI_EXPRESS_CORRECTABLE_ERROR_MASK	The PCI_EXPRESS_CORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) correctable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_CORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_CORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) correctable error status register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_DEVICE_CAPABILITIES_REGISTER	The PCI_EXPRESS_DEVICE_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) device capabilities register of a PCIe capability structure.
PCI_EXPRESS_DEVICE_CONTROL_REGISTER	The PCI_EXPRESS_DEVICE_CONTROL_REGISTER structure describes a PCI Express (PCIe) device control register of a PCIe capability structure.
PCI_EXPRESS_DEVICE_STATUS_REGISTER	The PCI_EXPRESS_DEVICE_STATUS_REGISTER structure describes a PCI Express (PCIe) device status register of a PCIe capability structure.
PCI_EXPRESS_ENHANCED_CAPABILITY_HEADER	The PCI_EXPRESS_ENHANCED_CAPABILITY_HEADER structure describes the header for a PCI Express (PCIe) extended capability structure.
PCI_EXPRESS_ERROR_SOURCE_ID	The PCI_EXPRESS_ERROR_SOURCE_ID structure describes the identifiers of the first correctable error and the first uncorrectable error that are reported in the PCI Express (PCIe) root error status register.
PCI_EXPRESS_LINK_CAPABILITIES_REGISTER	The PCI_EXPRESS_LINK_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) link capabilities register of a PCIe capability structure.

TITLE	DESCRIPTION
PCI_EXPRESS_LINK_CONTROL_REGISTER	The PCI_EXPRESS_LINK_CONTROL_REGISTER structure describes a PCI Express (PCIe) link control register of a PCIe capability structure.
PCI_EXPRESS_LINK_STATUS_REGISTER	The PCI_EXPRESS_LINK_STATUS_REGISTER structure describes a PCI Express (PCIe) link status register of a PCIe capability structure.
PCI_EXPRESS_PME_REQUESTOR_ID	The PCI_EXPRESS_PME_REQUESTOR_ID structure describes the identifier of the requester of a power management event (PME).
PCI_EXPRESS_ROOT_CAPABILITIES_REGISTER	The PCI_EXPRESS_ROOT_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) root capabilities register of a PCIe capability structure.
PCI_EXPRESS_ROOT_CONTROL_REGISTER	The PCI_EXPRESS_ROOT_CONTROL_REGISTER structure describes a PCI Express (PCIe) root control register of a PCIe capability structure.
PCI_EXPRESS_ROOT_ERROR_COMMAND	The PCI_EXPRESS_ROOT_ERROR_COMMAND structure describes a PCI Express (PCIe) root error command register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_ROOT_ERROR_STATUS	The PCI_EXPRESS_ROOT_ERROR_STATUS structure describes a PCI Express (PCIe) root error status register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_ROOT_STATUS_REGISTER	The PCI_EXPRESS_ROOT_STATUS_REGISTER structure describes a PCI Express (PCIe) root status register of a PCIe capability structure.
PCI_EXPRESS_ROOTPORT_AER_CAPABILITY	The PCI_EXPRESS_ROOTPORT_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure for a root port or a root complex event collector.
PCI_EXPRESS_SEC_AER_CAPABILITIES	The PCI_EXPRESS_SEC_AER_CAPABILITIES structure describes a PCI Express (PCIe) secondary error capabilities and control register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_MASK	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) secondary uncorrectable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_SEVERITY	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_SEVERITY structure describes a PCI Express (PCIe) secondary uncorrectable error severity register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) secondary uncorrectable error status register of a PCIe advanced error reporting capability structure.

TITLE	DESCRIPTION
PCI_EXPRESS_SERIAL_NUMBER_CAPABILITY	The PCI_EXPRESS_SERIAL_NUMBER_CAPABILITY structure describes a serial number for a PCI Express (PCIe) device.
PCI_EXPRESS_SLOT_CAPABILITIES_REGISTER	The PCI_EXPRESS_SLOT_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) slot capabilities register of a PCIe capability structure.
PCI_EXPRESS_SLOT_CONTROL_REGISTER	The PCI_EXPRESS_SLOT_CONTROL_REGISTER structure describes a PCI Express (PCIe) slot control register of a PCIe capability structure.
PCI_EXPRESS_SLOT_STATUS_REGISTER	The PCI_EXPRESS_SLOT_STATUS_REGISTER structure describes a PCI Express (PCIe) slot status register of a PCIe capability structure.
PCI_EXPRESS_UNCORRECTABLE_ERROR_MASK	The PCI_EXPRESS_UNCORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) uncorrectable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_UNCORRECTABLE_ERROR_SEVERITY	The PCI_EXPRESS_UNCORRECTABLE_ERROR_SEVERITY structure describes a PCI Express (PCIe) uncorrectable error severity register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_UNCORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_UNCORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) uncorrectable error status register of a PCIe advanced error reporting capability structure.
PCI_PM_CAPABILITY	The PCI_PM_CAPABILITY structure reports the power management capabilities of the device.
PCI_PMC	The PCI_PMC structure is used to report the contents of the power management capabilities register.
PCI_PMCSR	The PCI_PMCSR structure is used to report the contents of the device's power management control status register.
PCI_PMCSR_BSE	The PCI_PMCSR_BSE structure is used to report the contents of the power management control status register for PCI bridge support extensions.
PCI_SLOT_NUMBER	The PCI_SLOT_NUMBER structure is obsolete.
PCI_X_CAPABILITY	The PCI_X_CAPABILITY structure reports the contents of the command and status registers of a device that is compliant with the PCI-X Addendum to the PCI Local Bus Specification.
PROCESSOR_NUMBER	The PROCESSOR_NUMBER structure identifies a processor by its group number and group-relative processor number.

Enumerations

TITLE	DESCRIPTION
BUS_DATA_TYPE	The BUS_DATA_TYPE enumeration indicates the type of bus configuration space.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
IRQ_DEVICE_POLICY	The IRQ_DEVICE_POLICY enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.
IRQ_PRIORITY	The IRQ_PRIORITY enumeration type indicates the priority the system should give to servicing a device's interrupts.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
KINTERRUPT_POLARITY	The KINTERRUPT_POLARITY enumeration indicates how a device signals an interrupt request on an interrupt line.
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.

BUS_DATA_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **BUS_DATA_TYPE** enumeration indicates the type of bus configuration space.

Syntax

```
typedef enum _BUS_DATA_TYPE {
  ConfigurationSpaceUndefined,
  Cmos,
  EisaConfiguration,
  Pos,
  CbusConfiguration,
  PCIConfiguration,
  VMEConfiguration,
  NuBusConfiguration,
  PCMCIAConfiguration,
  MPIConfiguration,
  MPSAConfiguration,
  PNPISAConfiguration,
  SgiInternalConfiguration,
  MaximumBusDataType
} BUS_DATA_TYPE, *PBUS_DATA_TYPE;
```

Constants

ConfigurationSpaceUndefined	Indicates that the type of bus configuration space is undefined.
Cmos	Indicates CMOS data.
EisaConfiguration	Indicates an EISA bus configuration space.
Pos	For internal use only.
CbusConfiguration	Indicates Cbus configuration space.
PCIConfiguration	Indicates PCI configuration space.
VMEConfiguration	Indicates VME configuration space.
NuBusConfiguration	Indicates NuBus configuration space.
PCMCIAConfiguration	Indicates PCMCIA configuration space.
MPIConfiguration	Indicates MPI configuration space.
MPSAConfiguration	Indicates MPSA configuration space.
PNPISAConfiguration	Indicates PNPISA configuration space.

SgiInternalConfiguration	Indicates SGI internal bus configuration space.
MaximumBusDataType	Indicates the upper limit of the bus configuration space types.

Requirements

Header	miniport.h (include Ntddk.h, Miniport.h)

See Also

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

CM_EISA_FUNCTION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_EISA_FUNCTION_INFORMATION** structure defines detailed EISA configuration information returned by HalGetBusData for the input *BusDataType***EisaConfiguration**, or by **HalGetBusDataByOffset** for the input *BusDataType***EisaConfiguration** and the *Offset* zero, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _CM_EISA_FUNCTION_INFORMATION {
  ULONG                     CompressedId;
  UCHAR                     IdSlotFlags1;
  UCHAR                     IdSlotFlags2;
  UCHAR                     MinorRevision;
  UCHAR                     MajorRevision;
  UCHAR                     Selections[26];
  UCHAR                     FunctionFlags;
  UCHAR                     TypeString[80];
  EISA_MEMORY_CONFIGURATION EisaMemory[9];
  EISA_IRQ_CONFIGURATION    EisaIrq[7];
  EISA_DMA_CONFIGURATION    EisaDma[4];
  EISA_PORT_CONFIGURATION   EisaPort[20];
  UCHAR                     InitializationData[60];
} CM_EISA_FUNCTION_INFORMATION, *PCM_EISA_FUNCTION_INFORMATION;
```

Members

`CompressedId`

The EISA compressed identification of the device at this slot. The value is identical to the **CompressedId** member of the CM_EISA_SLOT_INFORMATION structure.

`IdSlotFlags1`

The EISA slot identification flags.

`IdSlotFlags2`

The EISA slot identification flags.

`MinorRevision`

Information supplied by the manufacturer.

`MajorRevision`

Information supplied by the manufacturer.

`Selections`

The EISA selections for the device.

`FunctionFlags`

Indicates which of the members has available information. Callers can use the following system-defined masks to determine whether a particular type of configuration information can be or has been returned by **HalGetBusData**

or HalGetBusDataByOffset:

EISA_FUNCTION_ENABLED

EISA_FREE_FORM_DATA

EISA_HAS_PORT_INIT_ENTRY

EISA_HAS_PORT_RANGE

EISA_HAS_DMA_ENTRY

EISA_HAS_IRQ_ENTRY

EISA_HAS_MEMORY_ENTRY

EISA_HAS_TYPE_ENTRY

EISA_HAS_INFORMATION

The EISA_HAS_INFORMATION mask is a combination of the following:

EISA_HAS_PORT_RANGE

EISA_HAS_DMA_ENTRY

EISA_HAS_IRQ_ENTRY

EISA_HAS_MEMORY_ENTRY

EISA_HAS_TYPE_ENTRY

`TypeString`

Specifies the type of device.

`EisaMemory`

Describes the EISA device memory configuration information, defined as follows:

```
typedef struct _EISA_MEMORY_CONFIGURATION {
    EISA_MEMORY_TYPE ConfigurationByte;
    UCHAR DataSize;
    USHORT AddressLowWord;
    UCHAR AddressHighByte;
    USHORT MemorySize;
} EISA_MEMORY_CONFIGURATION, *PEISA_MEMORY_CONFIGURATION;
```

`EisaIrq`

Describes the EISA interrupt configuration information, defined as follows:

```
typedef struct _EISA_IRQ_CONFIGURATION {
    EISA_IRQ_DESCRIPTOR ConfigurationByte;
    UCHAR Reserved;
} EISA_IRQ_CONFIGURATION, *PEISA_IRQ_CONFIGURATION;
```

`EisaDma`

Describes the EISA DMA configuration information, defined as follows:

```
typedef struct _EISA_DMA_CONFIGURATION {
    DMA_CONFIGURATION_BYTE0 ConfigurationByte0;
    DMA_CONFIGURATION_BYTE1 ConfigurationByte1;
} EISA_DMA_CONFIGURATION, *PEISA_DMA_CONFIGURATION;
```

`EisaPort`

Describes the EISA device port configuration information, defined as follows:

```
typedef struct _EISA_PORT_CONFIGURATION {
    EISA_PORT_DESCRIPTOR Configuration;
    USHORT PortAddress;
} EISA_PORT_CONFIGURATION, *PEISA_PORT_CONFIGURATION;
```

`InitializationData`

Vendor-supplied, device-specific initialization data, if any.

Remarks

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in **CM_EISA_FUNCTION_INFORMATION** and/or in the CM_EISA_SLOT_INFORMATION header immediately preceding it is read-only.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_EISA_SLOT_INFORMATION

HalGetBusData

HalGetBusDataByOffset

CM_EISA_SLOT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_EISA_SLOT_INFORMATION** structure defines EISA configuration header information returned by HalGetBusData for the input *BusDataType* = **EisaConfiguration**, or by HalGetBusDataByOffset for the inputs *BusDataType* = **EisaConfiguration** and *Offset* = 0, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _CM_EISA_SLOT_INFORMATION {
  UCHAR  ReturnCode;
  UCHAR  ReturnFlags;
  UCHAR  MajorRevision;
  UCHAR  MinorRevision;
  USHORT Checksum;
  UCHAR  NumberFunctions;
  UCHAR  FunctionInformation;
  ULONG  CompressedId;
} CM_EISA_SLOT_INFORMATION, *PCM_EISA_SLOT_INFORMATION;
```

Members

`ReturnCode`

Contains a status code if an error occurs when the EISA BIOS is queried. Possible status codes include the following:

- EISA_INVALID_SLOT
- EISA_INVALID_FUNCTION
- EISA_INVALID_CONFIGURATION
- EISA_EMPTY_SLOT
- EISA_INVALID_BIOS_CALL

`ReturnFlags`

The return flags.

`MajorRevision`

Information supplied by the manufacturer.

`MinorRevision`

Information supplied by the manufacturer.

`Checksum`

The checksum value, allowing validation of the configuration data.

`NumberFunctions`

The number at this slot.

`FunctionInformation`

Whether there is available CM_EISA_FUNCTION_INFORMATION for this slot.

`CompressedId`

The EISA compressed identification of the device at this slot. This value is identical to the **CompressedId** member of the **CM_EISA_FUNCTION_INFORMATION** structure. This member can be read to determine whether the caller should call **HalGetBusData** or **HalGetBusDataByOffset** again with sufficient buffer space to get more detailed **CM_EISA_FUNCTION_INFORMATION** for a device it supports.

Remarks

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in **CM_EISA_SLOT_INFORMATION** and in **CM_EISA_FUNCTION_INFORMATION** immediately following it is read-only.

The driver of an EISA device might call **HalGetBusData** or **HalGetBusDataByOffset** for each slot on each EISA bus in the system, requesting only **CM_EISA_SLOT_INFORMATION** in order to find the device(s) it supports by examining the returned **CompressedId** values. Then, such a driver could allocate sufficient buffer space to call **HalGetBusData(ByOffset)** again for **CM_EISA_SLOT_INFORMATION** and **CM_EISA_FUNCTION_INFORMATION** at slots where its device(s) can be found.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_EISA_FUNCTION_INFORMATION

HalGetBusData

HalGetBusDataByOffset

CM_MCA_POS_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_MCA_POS_DATA** structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.

Syntax

```
typedef struct _CM_MCA_POS_DATA {
  USHORT AdapterId;
  UCHAR  PosData1;
  UCHAR  PosData2;
  UCHAR  PosData3;
  UCHAR  PosData4;
} CM_MCA_POS_DATA, *PCM_MCA_POS_DATA;
```

Members

`AdapterId`

`PosData1`

`PosData2`

`PosData3`

`PosData4`

Remarks

This structure is used by the obsolete HalGetBusData and HalGetBusDataByOffset routines.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalGetBusData

HalGetBusDataByOffset

GROUP_AFFINITY structure

1/30/2019 • 2 minutes to read • Edit Online

The **GROUP_AFFINITY** structure specifies a group number and the processor affinity within that group.

Syntax

```
typedef struct _GROUP_AFFINITY {
  KAFFINITY Mask;
  USHORT    Group;
  USHORT    Reserved[3];
} GROUP_AFFINITY, *PGROUP_AFFINITY;
```

Members

`Mask`

Specifies the affinity mask. This parameter is a **KAFFINITY** value. The bits in the affinity mask identify a set of processors within the group identified by **Group**.

`Group`

Specifies the group number. In Windows 7, the group number must be in the range from 0 to 3.

`Reserved`

Reserved for future use. Set all three **Reserved** array elements to zero.

Remarks

This structure describes a group-specific affinity.

A **GROUP_AFFINITY** structure can describe a thread affinity, which is a set of processors on which a thread is allowed to run. All of the processors in this set belong to the group that is identified by the **Group** member of the structure. The **Mask** member contains an affinity mask that identifies the processors in the set. For example, the KeSetSystemGroupAffinityThread and KeRevertToUserGroupAffinityThread routines use **GROUP_AFFINITY** structures to specify thread affinities.

A **GROUP_AFFINITY** structure can describe an interrupt affinity, which is a set of processors on which an interrupt service routine can receive interrupts. For example, the IoGetDeviceNumaNode routine uses **GROUP_AFFINITY** structures to specify interrupt affinities.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	miniport.h (include Winnt.h, Ntdef.h, Windef.h, Minitape.h)

See Also

IoGetDeviceNumaNode

KAFFINITY

KeRevertToUserGroupAffinityThread

KeSetSystemGroupAffinityThread

INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **INTERFACE** structure describes an interface that is exported by a driver for use by other drivers.

Syntax

```
typedef struct _INTERFACE {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
} INTERFACE, *PINTERFACE;
```

Members

`Size`

Size, in bytes, of a structure defining a driver interface, including this structure and interface-specific members.

`Version`

Driver-defined interface version.

`Context`

Pointer to interface-specific context information.

`InterfaceReference`

Pointer to a driver-supplied InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

Pointer to a driver-supplied InterfaceDereference routine that decrements the interface's reference count.

Remarks

The **INTERFACE** structure must be included as the first member of all structures that describe interfaces returned by drivers in response to an IRP_MN_QUERY_INTERFACE request.

The *InterfaceReference* routine must be called by the driver that exports the interface, each time the driver supplies that interface in response to **IRP_MN_QUERY_INTERFACE**. Likewise, if the driver that requests the interface subsequently passes it to another driver, the driver that passes the interface must call *InterfaceReference* on behalf of the driver that receives it.

Each driver that imports the interface (whether by sending **IRP_MN_QUERY_INTERFACE** or by receiving the interface from another driver) must call the *InterfaceDereference* routine after it has finished using the interface. After calling the *InterfaceDereference* routine, a driver cannot use the interface again without first reobtaining it.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

INTERFACE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **INTERFACE_TYPE** enumeration indicates the bus type.

Syntax

```
typedef enum _INTERFACE_TYPE {
  InterfaceTypeUndefined,
  Internal,
  Isa,
  Eisa,
  MicroChannel,
  TurboChannel,
  PCIBus,
  VMEBus,
  NuBus,
  PCMCIABus,
  CBus,
  MPIBus,
  MPSABus,
  ProcessorInternal,
  InternalPowerBus,
  PNPISABus,
  PNPBus,
  Vmcs,
  ACPIBus,
  MaximumInterfaceType
} INTERFACE_TYPE, *PINTERFACE_TYPE;
```

Constants

InterfaceTypeUndefined	Indicates that the interface type is undefined.
Internal	For internal use only.
Isa	Indicates that the interface is published by the ISA bus driver.
Eisa	Indicates that the interface is published by the EISA bus driver.
MicroChannel	Indicates that the interface is published by the MicroChannel bus driver.
TurboChannel	Indicates that the interface is published by the TurboChannel bus driver.
PCIBus	Indicates that the interface is published by the PCI bus driver.
VMEBus	Indicates that the interface is published by the VME bus driver.
NuBus	Indicates that the interface is published by the NuBus driver.

PCMCIABus	Indicates that the interface is published by the PCMCIA bus driver.
CBus	Indicates that the interface is published by the Cbus driver.
MPIBus	Indicates that the interface is published by the MPI bus driver.
MPSABus	Indicates that the interface is published by the MPSA bus driver.
ProcessorInternal	Indicates that the interface is published by the ISA bus driver.
InternalPowerBus	Indicates that the interface is published for an internal power bus. Some devices have power control ports that allow them to share power control with other devices. The Windows architecture represents these devices as slots on a virtual bus called an "internal power bus."
PNPISABus	Indicates that the interface is published by the PNPISA bus driver.
PNPBus	Indicates that the interface is published by the PNP bus driver.
Vmcs	Reserved for use by the operating system.
ACPIBus	Indicates that the interface is published by the ACPI bus driver. The ACPI bus driver enumerates devices that are described in the ACPI firmware of the hardware platform. These devices might physically reside on buses that are controlled by other bus drivers, but the ACPI bus driver must enumerate these devices because the other bus drivers cannot detect them. This interface type is defined starting with Windows 8.
MaximumInterfaceType	Marks the upper limit of the possible bus types.

Requirements

Header	miniport.h (include Wdm.h, Miniport.h, Wudfwdm.h)

See Also

HW_INITIALIZATION_DATA

PORT_CONFIGURATION_INFORMATION

ScsiPortGetDeviceBase

ScsiPortValidateRange

StorPortGetDeviceBase

StorPortValidateRange

IO_RESOURCE_DESCRIPTOR structure

1/30/2019 • 7 minutes to read • Edit Online

The **IO_RESOURCE_DESCRIPTOR** structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of **IO_RESOURCE_DESCRIPTOR** structures is contained within each IO_RESOURCE_LIST structure.

Syntax

```
typedef struct _IO_RESOURCE_DESCRIPTOR {
  UCHAR  Option;
  UCHAR  Type;
  UCHAR  ShareDisposition;
  UCHAR  Spare1;
  USHORT Flags;
  USHORT Spare2;
  union {
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Port;
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory;
    struct {
      ULONG             MinimumVector;
      ULONG             MaximumVector;
      IRQ_DEVICE_POLICY AffinityPolicy;
      USHORT            Group;
      IRQ_PRIORITY      PriorityPolicy;
      KAFFINITY         TargetedProcessors;
    } Interrupt;
    struct {
      ULONG MinimumChannel;
      ULONG MaximumChannel;
    } Dma;
    struct {
      ULONG RequestLine;
      ULONG Reserved;
      ULONG Channel;
      ULONG TransferWidth;
    } DmaV3;
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Generic;
    struct {
      ULONG Data[3];
    } DevicePrivate;
    struct {
      ULONG Length;
      ULONG MinBusNumber;
      ULONG MaxBusNumber;
      ULONG Reserved;
```

```
      ULONG Reserved;
    } BusNumber;
    struct {
      ULONG Priority;
      ULONG Reserved1;
      ULONG Reserved2;
    } ConfigData;
    struct {
      ULONG            Length40;
      ULONG            Alignment40;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory40;
    struct {
      ULONG            Length48;
      ULONG            Alignment48;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory48;
    struct {
      ULONG            Length64;
      ULONG            Alignment64;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory64;
    struct {
      UCHAR Class;
      UCHAR Type;
      UCHAR Reserved1;
      UCHAR Reserved2;
      ULONG IdLowPart;
      ULONG IdHighPart;
    } Connection;
  } u;
} IO_RESOURCE_DESCRIPTOR, *PIO_RESOURCE_DESCRIPTOR;
```

Members

`Option`

Specifies whether this resource description is required, preferred, or alternative. One of the following values must be used:

VALUE	MEANING
0	The specified resource range is required, unless alternative ranges are also specified.
IO_RESOURCE_PREFERRED	The specified resource range is preferred to any alternative ranges.

IO_RESOURCE_ALTERNATIVE	The specified resource range is an alternative to the range preceding it. For example, if one **IO_RESOURCE_DESCRIPTOR** structure specifies IRQ 5, with IO_RESOURCE_PREFERRED set, and the next structure specifies IRQ 3, with IO_RESOURCE_ALTERNATIVE set, the PnP manager assigns IRQ 3 to the device only if IRQ 5 is unavailable. (Multiple alternatives can be specified for each resource. Both IO_RESOURCE_ALTERNATIVE and IO_RESOURCE_PREFERRED can be set, indicating a preferred alternative.)
IO_RESOURCE_DEFAULT	Not used.

`Type`

Identifies the resource type. For a list of valid values, see the **Type** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`ShareDisposition`

Indicates whether the described resource can be shared. For a list of valid values, see the **ShareDisposition** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`Spare1`

`Flags`

Contains bit flags that are specific to the resource type. The following table shows the flags that are valid if **Type** = **CmResourceTypeInterrupt.**

VALUE	MEANING
CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE	The IRQ line is level-triggered. (These IRQs are usually sharable.)

CM_RESOURCE_INTERRUPT_LATCHED	The IRQ line is edge-triggered.
CM_RESOURCE_INTERRUPT_MESSAGE	If this flag is set, the interrupt is a message-signaled interrupt. Otherwise, the interrupt is a line-based interrupt. This flag can be set starting with Windows Vista.
CM_RESOURCE_INTERRUPT_POLICY_INCLUDED	If this flag is set, the u.Interrupt member includes data that describes the device's interrupt policy. This flag can be set starting with Windows Vista.
CM_RESOURCE_INTERRUPT_SECONDARY_INTERRUPT	The interrupt is a secondary interrupt. This flag can be set starting with Windows 8. For more information about secondary interrupts, see GPIO Interrupts.

CM_ RES OU RCE _IN TER RUP T_W AKE _HI NT	The interrupt is capable of waking the operating system from a low-power idle state or a system sleep state. This flag can be set starting with Windows 8. For more information about wake capabilities, see Enabling Device Wake-Up.

For a list of valid flags for other resource types, see the description of the **Flags** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`Spare2`

`u`

port

Specifies a range of I/O port addresses, using the following members.

Drivers for Windows Vista and later versions of the Windows operating system must use RtlIoDecodeMemIoResource and RtlIoEncodeMemIoResource to read and update this member, rather than updating it directly.

Length

The length, in bytes, of the range of assignable I/O port addresses.

Alignment

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be an integer multiple of *Alignment*.

MinimumAddress

The minimum bus-relative I/O port address that can be assigned to the device.

MaximumAddress

The maximum bus-relative I/O port address that can be assigned to the device.

`u.Port`

`u.Port.Length`

`u.Port.Alignment`

`u.Port.MinimumAddress`

`u.Port.MaximumAddress`

`u.Memory`

`u.Memory.Length`

The length, in bytes, of the range of assignable memory addresses.

`u.Memory.Alignment`

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be an integer multiple of *Alignment*.

`u.Memory.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Interrupt`

`u.Interrupt.MinimumVector`

The minimum bus-relative vector that can be assigned to the device.

`u.Interrupt.MaximumVector`

The maximum bus-relative vector that can be assigned to the device.

If the **CM_RESOURCE_INTERRUPT_MESSAGE** flag bit is set, the values of the **MinimumVector** and **MaximumVector** members have special meanings. For more information, see Using Interrupt Resource Descriptors.

The system supplies the following additional members on Windows Vista and later versions of the Windows operating system:

`u.Interrupt.AffinityPolicy`

Specifies an IRQ_DEVICE_POLICY value that indicates how the system should distribute a device's interrupts between processors.

Specifies an IRQ_DEVICE_POLICY value that indicates how the system should distribute a device's interrupts between processors.

`u.Interrupt.Group`

Specifies a processor group number. **Group** is a valid (but optional) member of **u.Interrupt** only in Windows 7 and later versions of Windows. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. If the **Group** member exists, the **Group** and **TargetedProcessors** members together specify a group affinity that identifies the set of processors that should handle the device's interrupts. To specify an affinity for a particular group, set **AffinityPolicy** to **IrqPolicySpecifiedProcessors** and set **Group** to the appropriate group number. In addition, **TargetedProcessors** must specify the target processors in the group. If you set **AffinityPolicy** to a value other than **IrqPolicySpecifiedProcessors**, set **Group** to ALL_PROCESSOR_GROUPS to indicate that the driver is group-aware (that is, designed to handle information about processor groups). A driver cannot specify target processors if **Group** equals ALL_PROCESSOR_GROUPS; such target specifications are ignored.

`u.Interrupt.PriorityPolicy`

Specifies an IRQ_PRIORITY value that indicates the priority with which the system should dispatch the device's interrupts.

`u.Interrupt.TargetedProcessors`

Specifies a **KAFFINITY** value that indicates which processors should handle the device's interrupts. This value is used only if **AffinityPolicy** is **IrqPolicySpecifiedProcessors.**

`u.Dma`

`u.Dma.MinimumChannel`

The minimum bus-relative DMA channel that can be assigned to the device.

`u.Dma.MaximumChannel`

The maximum bus-relative DMA channel that can be assigned to the device.

`u.DmaV3`

`u.DmaV3.RequestLine`

The number of the request line on the system DMA controller that is allocated to the device.

`u.DmaV3.Reserved`

Not used.

`u.DmaV3.Channel`

The number of the DMA channel on the system DMA controller that is allocated to the device.

`u.DmaV3.TransferWidth`

Specifies the width, in bits, of the data bus that the system DMA controller that is allocated to the device uses to transfer data to or from the device.

`u.Generic`

`u.Generic.Length`

`u.Generic.Alignment`

`u.Generic.MinimumAddress`

`u.Generic.MaximumAddress`

`u.DevicePrivate`

`u.DevicePrivate.Data`

`u.BusNumber`

`u.BusNumber.Length`

The number of bus numbers required.

`u.BusNumber.MinBusNumber`

The minimum bus-relative bus number that can be assigned to the device.

`u.BusNumber.MaxBusNumber`

The maximum bus-relative bus number that can be assigned to the device.

`u.BusNumber.Reserved`

Not used.

`u.ConfigData`

`u.ConfigData.Priority`

`u.ConfigData.Reserved1`

`u.ConfigData.Reserved2`

`u.Memory40`

`u.Memory40.Length40`

The high 32 bits of the 40-bit length, in bytes, of the range of assignable memory addresses. The lower 8 bits are treated as zero.

`u.Memory40.Alignment40`

The high 32 bits of the 40-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 8 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory40.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory40.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Memory48`

`u.Memory48.Length48`

The high 32 bits of the 48-bit length, in bytes, of the range of assignable memory addresses. The lower 16 bits are treated as zero.

`u.Memory48.Alignment48`

The high 32 bits of the 48-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 16 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory48.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory48.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Memory64`

`u.Memory64.Length64`

The high 32 bits of the 64-bit length, in bytes, of the range of assignable memory addresses. The lower 32 bits are treated as zero.

`u.Memory64.Alignment64`

The high 32 bits of the 64-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 32 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory64.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory64.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Connection`

`u.Connection.Class`

The connection class. This member is set to one of the following values.

VALUE	MEANING

CM_RESOURCE_CONNECTION_CLASS_GPIO	Access the device through one or more pins on a GPIO controller.
CM_RESOURCE_CONNECTION_CLASS_SERIAL	Access the device through a serial bus or serial port.

`u.Connection.Type`

The connection type. If *Class* = CM_RESOURCE_CONNECTION_CLASS_GPIO, *Type* is set to the following value:

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_GPIO_IO	Access the device through GPIO pins that are configured for I/O.

Note A GPIO pin that is configured as an interrupt request input is accessed as an ordinary interrupt resource (**CmResourceTypeInterrupt**).

If **Class** = CM_RESOURCE_CONNECTION_CLASS_SERIAL, **Type** is set to one of the following values:

VALUE	MEANING

CM_RESOURCE_CONNECTION_TYPE_I2C	The device is connected to an I²C bus.
CM_RESOURCE_CONNECTION_TYPE_SPI	The device is connected to an SPI bus.
CM_RESOURCE_CONNECTION_TYPE_SERIAL_UART	The device is connected to a serial port.

`u.Connection.Reserved1`

Not used.

`u.Connection.Reserved2`

Not used.

`u.Connection.IdLowPart`

The lower 32 bits of the 64-bit connection ID.

`u.Connection.IdHighPart`

The upper 32 bits of the 64-bit connection ID.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IO_RESOURCE_LIST

IO_RESOURCE_REQUIREMENTS_LIST

IoConnectInterrupt

IO_RESOURCE_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_RESOURCE_LIST** structure describes a range of raw hardware resources, of various types, that can be used by a device. The resources specified represent a single, acceptable resource configuration for a device. An array of **IO_RESOURCE_LIST** structures is contained within each IO_RESOURCE_REQUIREMENTS_LIST structure.

Syntax

```
typedef struct _IO_RESOURCE_LIST {
  USHORT                  Version;
  USHORT                  Revision;
  ULONG                   Count;
  IO_RESOURCE_DESCRIPTOR Descriptors[1];
} IO_RESOURCE_LIST, *PIO_RESOURCE_LIST;
```

Members

`Version`

The version number of this structure. This value should be 1.

`Revision`

The revision of this structure. This value should be 1.

`Count`

The number of elements in the **Descriptors** array.

`Descriptors`

The first element in an array of one or more IO_RESOURCE_DESCRIPTOR structures.

Remarks

This structure is the header for an array of **IO_RESOURCE_DESCRIPTOR** structures. The **Descriptors** member contains the first element in this array, and the **Count** member specifies the total number of array elements. If the array contains more than one element, the remaining elements in the array immediately follow the **IO_RESOURCE_LIST** structure in memory. The total number of bytes occupied by the **IO_RESOURCE_LIST** structure and any array elements that follow this structure is **sizeof**(**IO_RESOURCE_LIST**) + (**Count** - 1) * **sizeof**(**IO_RESOURCE_DESCRIPTOR**).

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IO_RESOURCE_REQUIREMENTS_LIST

IO_RESOURCE_REQUIREMENTS_LIST structure

1/30/2019 • 3 minutes to read • Edit Online

The **IO_RESOURCE_REQUIREMENTS_LIST** structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.

Syntax

```
typedef struct _IO_RESOURCE_REQUIREMENTS_LIST {
  ULONG            ListSize;
  INTERFACE_TYPE   InterfaceType;
  ULONG            BusNumber;
  ULONG            SlotNumber;
  ULONG            Reserved[3];
  ULONG            AlternativeLists;
  IO_RESOURCE_LIST List[1];
} IO_RESOURCE_REQUIREMENTS_LIST, *PIO_RESOURCE_REQUIREMENTS_LIST;
```

Members

`ListSize`

The total number of bytes that constitute the **IO_RESOURCE_REQUIREMENTS_LIST** structure, its IO_RESOURCE_LIST array, and the latter's IO_RESOURCE_DESCRIPTOR array.

`InterfaceType`

Specifies an interface type. This must be one of the types defined by INTERFACE_TYPE, in Wdm.h or Ntddk.h. (Not used by WDM drivers.)

`BusNumber`

A system-assigned, zero-based bus number. (Not used by WDM drivers.)

`SlotNumber`

A system slot number. (Not used by WDM drivers.)

`Reserved`

Not used.

`AlternativeLists`

The number of alternative lists (or configurations) of hardware resources that are described by this **IO_RESOURCE_REQUIREMENTS_LIST** structure. The **List** member is the header for the first alternative list.

`List`

The IO_RESOURCE_LIST structure that serves as the header for the first alternative list of hardware resources. If the **IO_RESOURCE_REQUIREMENTS_LIST** structure describes more than one alternative list, the second alternative list immediately follows the first alternative list in memory, and so on. The size of each alternative list depends on the length of the IO_RESOURCE_DESCRIPTOR array that it contains. For more information, see the following Remarks section.

Remarks

This structure is used by the IRP_MN_FILTER_RESOURCE_REQUIREMENTS and IRP_MN_QUERY_RESOURCE_REQUIREMENTS IRPs to describe one or more alternative configurations of hardware resources.

The **IO_RESOURCE_REQUIREMENTS_LIST** structure is a header for a larger data structure, of variable size, that specifies one or more alternative lists of resources. All of the data in this larger structure occupies a contiguous block of memory, whose size is specified by the **ListSize** member. Each alternative list occupies a subblock within the larger block.

Each alternative list begins with an **IO_RESOURCE_LIST** structure, which serves as a header for an array of **IO_RESOURCE_DESCRIPTOR** structures. The length of this array determines the size of the list, which can vary from one alternative list to the next. Driver code can use pointer arithmetic to step from one alternative list to the next. For example, if a parameter named *list* is a pointer to the **IO_RESOURCE_LIST** structure at the start of one alternative list, *list* can be updated to point to the start of the next list as follows:

```
list = (PIO_RESOURCE_LIST)(list->Descriptors + list->Count);
```

In this example, `list->Descriptors` is a pointer to the start of the **IO_RESOURCE_DESCRIPTOR** array, and `list->Count` is the number of elements in the array. For more information about the **Descriptors** and **Count** members, see IO_RESOURCE_LIST. #### Examples If a driver contains a handler for an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** IRP, the handler typically walks the resource list that is pointed to by the **IoStatus.Information** field in the IRP. The following code example contains a function—named FilterResources—that is called in the handler to walk the list. This function either verifies that a particular resource is specified in the list, or changes the resource requirements in the list without changing the size of the list. (A function that changes the size of the list would need to allocate a buffer for the updated list, write the list to this buffer, update the **IoStatus.Information** field in the IRP to point to the updated list, and free the buffer that contains the original list.) The FilterResources function returns **TRUE** (probably from the **switch** statement, although the details are omitted to simplify the code example) if it finds the resource that it is looking for in the list, or if it makes a change to the list that does not change the size of the list. Otherwise, it returns **FALSE**.

```
/* Modify the resource list in place (without changing its size). */

BOOLEAN FilterResources(PIO_RESOURCE_REQUIREMENTS_LIST reqlist)
{
    PIO_RESOURCE_LIST list;

    list = reqlist->List;

    for (int ix = 0; ix < reqlist->AlternativeLists; ++ix)
    {
        /* Process resources in IO_RESOURCE_LIST block number ix. */

        for (int jx = 0; jx < list->Count; ++jx)
        {
            PIO_RESOURCE_DESCRIPTOR desc;

            desc = list->Descriptors + jx;

            switch (desc->Type)
            {
               /* Process element jx in Descriptors array. */

               ...

            }
        }

        /* Advance to next IO_RESOURCE_LIST block in memory. */

        list = (PIO_RESOURCE_LIST)(list->Descriptors + list->Count);
    }

    return FALSE;
}
```

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IO_RESOURCE_LIST

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

IRP_MN_QUERY_RESOURCE_REQUIREMENTS

IRQ_DEVICE_POLICY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IRQ_DEVICE_POLICY** enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.

Syntax

```
typedef enum _IRQ_DEVICE_POLICY {
  IrqPolicyMachineDefault,
  IrqPolicyAllCloseProcessors,
  IrqPolicyOneCloseProcessor,
  IrqPolicyAllProcessorsInMachine,
  IrqPolicySpecifiedProcessors,
  IrqPolicySpreadMessagesAcrossAllProcessors,
  IrqPolicyAllProcessorsInMachineWhenSteered
} IRQ_DEVICE_POLICY, *PIRQ_DEVICE_POLICY;
```

Constants

IrqPolicyMachineDefault	The device does not require any particular assignment of interrupts to processors.
IrqPolicyAllCloseProcessors	The operating system should assign interrupts from the device to processors that are close to the device. On non-NUMA computers, the effect of this value is identical to that of **IrqPolicyAllProcessorsInMachine**.
IrqPolicyOneCloseProcessor	The operating system should assign a single interrupt for the device to one processor that is close to the device. On non-NUMA computers, the operating system can assign the interrupt to any processor.
IrqPolicyAllProcessorsInMachine	The operating system should assign interrupts from the device to all processors.
IrqPolicySpecifiedProcessors	The operating system should assign interrupts from the device to a specific set of processors.
IrqPolicySpreadMessagesAcrossAllProcessors	The operating system should assign different message-signaled interrupts to different processors, if possible.
IrqPolicyAllProcessorsInMachineWhenSteered	

Remarks

The **Interrupt.AffinityPolicy** member of the IO_RESOURCE_DESCRIPTOR structure is an **IRQ_DEVICE_POLICY** enumeration value.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IRQ_PRIORITY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IRQ_PRIORITY** enumeration type indicates the priority the system should give to servicing a device's interrupts.

Syntax

```
typedef enum _IRQ_PRIORITY {
  IrqPriorityUndefined,
  IrqPriorityLow,
  IrqPriorityNormal,
  IrqPriorityHigh
} IRQ_PRIORITY, *PIRQ_PRIORITY;
```

Constants

IrqPriorityUndefined	Specifies that the device does not require any particular priority for its interrupts.
IrqPriorityLow	Specifies that the device's interrupts are of low priority. This setting is appropiate for devices that can tolerate higher-than-normal latency.
IrqPriorityNormal	Specifies that the device's interrupts are of normal priority.
IrqPriorityHigh	Specifies that the device's interrupts are of high priority. This setting is appropriate for devices that require low latency.

Remarks

The system uses **IRQ_PRIORITY** to assign IRQLs for devices. For example, it might assign a higher IRQL to a device that has an **IRQ_PRIORITY** of **IrqPriorityHigh** than it does to a device that has an **IRQ_PRIORITY** of **IrqPriorityLow**.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

KINTERRUPT_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KINTERRUPT_MODE** enumeration type indicates whether an interrupt is level-triggered or edge-triggered.

Syntax

```
typedef enum _KINTERRUPT_MODE {
  LevelSensitive,
  Latched
} KINTERRUPT_MODE;
```

Constants

LevelSensitive	The interrupt is level-triggered. This is the mode for traditional PCI line-based interrupts.
Latched	The interrupt is edge-triggered. This is the mode for PCI message-signaled interrupts.

Remarks

The interrupt mode is an electrical characteristic of the interrupt. A device signals a level-triggered interrupt by changing the voltage on the interrupt pin, and holding it there until the processor signals that the interrupt is received. In contrast, to signal an edge-triggered interrupt, the device changes the voltage state for a fixed period of time, before reverting to the original voltage state.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

IoConnectInterrupt

IoConnectInterruptEx

KINTERRUPT_POLARITY

KINTERRUPT_POLARITY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KINTERRUPT_POLARITY** enumeration indicates how a device signals an interrupt request on an interrupt line.

Syntax

```
typedef enum _KINTERRUPT_POLARITY {
  InterruptPolarityUnknown,
  InterruptActiveHigh,
  InterruptRisingEdge,
  InterruptActiveLow,
  InterruptFallingEdge,
  InterruptActiveBoth,
  InterruptActiveBothTriggerLow,
  InterruptActiveBothTriggerHigh
} KINTERRUPT_POLARITY, *PKINTERRUPT_POLARITY;
```

Constants

InterruptPolarityUnknown	The interrupt polarity is unknown.
InterruptActiveHigh	Active-high interrupt. The interrupt input type is level-triggered, and an interrupt request is indicated by a high signal level on the interrupt line. The request remains active as long as the line remains high.
InterruptRisingEdge	Rising-edge-triggered interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a low-to-high transition on the interrupt line.
InterruptActiveLow	Active-low interrupt. The interrupt input type is level-triggered, and an interrupt request is indicated by a low signal level on the interrupt line. The request remains active as long as the line remains low.
InterruptFallingEdge	Falling-edge-triggered interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a high-to-low transition on the interrupt line.
InterruptActiveBoth	Active-both interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a low-to-high or a high-to-low transition on the interrupt line. After a low-to-high transition signals an interrupt request, the interrupt line remains high until a high-to-low transition signals the next interrupt request. Similarly, after a high-to-low transition signals an interrupt request, the interrupt line remains low until a low-to-high transition signals the next interrupt request.
InterruptActiveBothTriggerLow	Reserved for use by the operating system.

InterruptActiveBothTriggerHigh	Reserved for use by the operating system.

Remarks

A **KINTERRUPT_POLARITY** enumeration constant is frequently used in conjunction with a KINTERRUPT_MODE enumeration constant to describe an interrupt signal. A **KINTERRUPT_MODE** enumeration constant indicates whether the interrupt signal from a device is level-triggered or edge-triggered.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

KINTERRUPT_MODE

MEMORY_CACHING_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **MEMORY_CACHING_TYPE** enumeration type specifies the permitted caching behavior when allocating or mapping memory.

Syntax

```
typedef enum _MEMORY_CACHING_TYPE {
  MmNonCached,
  MmCached,
  MmWriteCombined,
  MmHardwareCoherentCached,
  MmNonCachedUnordered,
  MmUSWCCached,
  MmMaximumCacheType,
  MmNotMapped
} MEMORY_CACHING_TYPE;
```

Constants

MmNonCached	The requested memory should not be cached by the processor.
MmCached	The processor should cache the requested memory.
MmWriteCombined	The requested memory should not be cached by the processor, but writes to the memory can be combined by the processor.
MmHardwareCoherentCached	Reserved for system use.
MmNonCachedUnordered	Reserved for system use.
MmUSWCCached	Reserved for system use.
MmMaximumCacheType	Reserved for system use.
MmNotMapped	

Remarks

Processor translation buffers cache virtual to physical address translations. These translation buffers allow many virtual addresses to map a single physical address. However, only one caching behavior is allowed for any given physical address translation. Therefore, if a driver maps two different virtual address ranges to the same physical address, it must ensure that it specifies the same caching behavior for both. Otherwise, the processor behavior is undefined with unpredictable system results.

Requirements

Minimum supported client	Supported in Windows 2000 and later versions of Windows.
Header	miniport.h (include Ntddk.h, Wdm.h, Miniport.h, Wudfwdm.h)

See Also

MmAllocateContiguousMemorySpecifyCache

MmFreeContiguousMemorySpecifyCache

MmMapIoSpace

MmMapLockedPagesSpecifyCache

MmMapLockedPagesWithReservedMapping

PCI_COMMON_CONFIG structure

1/30/2019 • 2 minutes to read • Edit Online

The **PCI_COMMON_CONFIG** structure is obsolete. It defines standard PCI configuration information returned by the obsolete HalGetBusData or HalGetBusDataByOffset routine for the input *BusDataType* PCIConfiguration, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _PCI_COMMON_CONFIG {
  UCHAR      DeviceSpecific[192];
  base_class PCI_COMMON_HEADER;
} PCI_COMMON_CONFIG, *PPCI_COMMON_CONFIG;
```

Members

`DeviceSpecific`

Contains any device-specific initialization information that is available.

`PCI_COMMON_HEADER`

Remarks

Certain members of this structure have read-only values, so attempts to reset them are ignored. These members include the following: **VendorID**, **DeviceID**, **RevisionID**, **ProgIf**, **SubClass**, **BaseClass**, **HeaderType**, **InterruptPin**, **MinimumGrant**, and **MaximumLatency.**

Other members are provisionally read-only: that is, the system initializes them to their correct values, so drivers can safely treat them as read-only. However, they can be reset if a bus-master driver finds it necessary. These members include the following: **CacheLineSize** and **LatencyTimer**.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

PCI_SLOT_NUMBER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PCI_SLOT_NUMBER** structure is obsolete. It defines the format of the *Slot* parameter to the obsolete **Hal*Xxx* BusData** routines when they are called with the *BusDataType* value **PCIConfiguration**.

Syntax

```
typedef struct _PCI_SLOT_NUMBER {
  union {
    struct {
      ULONG DeviceNumber : 5;
      ULONG FunctionNumber : 3;
      ULONG Reserved : 24;
    } bits;
    ULONG AsULONG;
  } u;
} PCI_SLOT_NUMBER, *PPCI_SLOT_NUMBER;
```

Members

`u`

`u.bits`

`u.bits.DeviceNumber`

`u.bits.FunctionNumber`

`u.bits.Reserved`

`u.AsULONG`

Specifies the logical slot number of the device being configured.

Remarks

Drivers of PCI devices can call HalGetBusData or HalGetBusDataByOffset more than once for the same slot number to get the configuration information for their device(s).

For example, a driver might search for devices it supports on all PCI buses in the machine first, and then call `HalGetBusData(ByOffset)` again to request more configuration information about devices of interest. Such a driver could code a loop that calls `HalGetBusData(ByOffset)` with an input `Buffer` of sufficient `Length` only to contain enough of the PCI_COMMON_CONFIG to determine the `VendorID` and `DeviceID` of each PCI device. After finding the `Slot` numbers for any promising PCI devices, the driver would call HalGetBusData or HalGetBusDataByOffset one or more times with additional buffer space to get the information needed to configure its devices.

Requirements

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

PCI_COMMON_CONFIG

PROCESSOR_NUMBER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PROCESSOR_NUMBER** structure identifies a processor by its group number and group-relative processor number.

Syntax

```
typedef struct _PROCESSOR_NUMBER {
  USHORT Group;
  UCHAR  Number;
  UCHAR  Reserved;
} PROCESSOR_NUMBER, *PPROCESSOR_NUMBER;
```

Members

`Group`

The group number. If multiprocessor system contains *n* groups, the groups are numbered from 0 to *n*-1.

`Number`

The group-relative processor number. If a group contains *m* logical processors, the processors are numbered from 0 to *m*-1.

`Reserved`

Reserved for future use. Initialize to zero.

Remarks

Windows 7 is the first version of Windows to support processor groups. In Windows 7, only 64-bit versions of Windows support multiple groups; 32-bit versions of Windows support only one group. If a multiprocessor system is running a 64-bit version of Windows and contains no more than 64 logical processors, Windows assigns all processors to group 0. A multiprocessor system that is running a 32-bit version of Windows can contain no more than 32 processors.

In Windows 7, the maximum number of groups in a multiprocessor system is four, but this value might change in future versions of Windows. The safest way to determine the maximum number of groups in Windows 7 or a later version of Windows is to call the KeQueryMaximumGroupCount routine. Kernel-mode drivers that call **KeQueryMaximumGroupCount** will not require code changes if the formula that is used to calculate the maximum number of groups changes in a future version of Windows.

Kernel-mode routines that use the **PROCESSOR_NUMBER** structure include KeGetCurrentProcessorNumberEx, KeGetProcessorIndexFromNumber, KeGetProcessorNumberFromIndex, and KeQueryLogicalProcessorRelationship.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	miniport.h (include Windows.h, Ntdef.h, Wdm.h, Ntddk.h, Winnt.h, Minitape.h)

See Also

KeGetCurrentProcessorNumberEx

KeGetProcessorIndexFromNumber

KeGetProcessorNumberFromIndex

KeQueryLogicalProcessorRelationship

KeQueryMaximumGroupCount

FIELD_OFFSET macro

1/25/2019 • 2 minutes to read • Edit Online

The **FIELD_OFFSET** macro returns the byte offset of a named field in a known structure type.

Syntax

```
void FIELD_OFFSET(
   type,
   field
);
```

Parameters

`type`

Specifies the name of a known structure type containing *Field*.

`field`

Specifies the name of a field in a structure of type *Type*.

Return Value

Returns the byte offset of the caller supplied Field in the Type structure.

Remarks

Used by device driver writers to symbolically determine the offset of a known field in a known structure type.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Miniport.h, Minitape.h, Scsi.h, Storport.h)
IRQL	Any level

See Also

CONTAINING_RECORD

InterlockedAnd function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedAnd** macro atomically computes a bitwise AND operation.

Syntax

```
LONG InterlockedAnd(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be ANDed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be ANDed with the variable that is pointed to by *Destination*.

Return Value

InterlockedAnd returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedAnd atomically computes ***Destination*&=*Value***.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedOr

InterlockedXor

InterlockedCompareExchange function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedCompareExchange** routine performs an atomic operation that compares the input value pointed to by *Destination* with the value of *Comparand*.

Syntax

```
LONG InterlockedCompareExchange(
  LONG volatile *Destination,
  LONG          ExChange,
  LONG          Comperand
);
```

Parameters

`Destination`

A pointer to the input value that is compared with the value of *Comparand*.

`ExChange`

Specifies the output value pointed to by *Destination* if the input value pointed to by *Destination* equals the value of *Comparand*.

`Comperand`

Specifies the value that is compared with the input value pointed to by *Destination*.

Return Value

InterlockedCompareExchange returns the original value of **Destination*.

Remarks

If *Comparand* is equal to **Destination*, then **Destination* is set to equal *Exchange*. Otherwise, **Destination* is unchanged.

InterlockedCompareExchange provides a fast, atomic way to synchronize the testing and updating of a variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comparand*, the output value of *Destination* is set to the value of *Exchange*.

InterlockedCompareExchange is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchange** is atomic only with respect to other **Interlocked*Xxx*** calls. It does not use a spin lock and can be safely used on pageable data.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedCompareExchange64

InterlockedCompareExchangePointer

InterlockedDecrement

InterlockedExchange

InterlockedExchangePointer

InterlockedIncrement

InterlockedCompareExchangePointer function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedCompareExchangePointer** routine performs an atomic operation that compares the input pointer value pointed to by *Destination* with the pointer value *Comparand*.

Syntax

```
PVOID InterlockedCompareExchangePointer(
  PVOID volatile *Destination,
  PVOID          Exchange,
  PVOID          Comperand
);
```

Parameters

`Destination`

A pointer to a PVOID value. If (*Destination) = Comparand, then the routine sets (Destination*) to *Exchange*.

`Exchange`

Specifies the PVOID value to set (**Destination*) to.

`Comperand`

Specifies the PVOID value to compare with (**Destination*).

Return Value

InterlockedCompareExchangePointer returns the original value of the pointer at **Destination* (that is, the value of this pointer at entry to the routine).

Remarks

If *Comparand* is equal to **Destination*, then **Destination* is set to equal *Exchange*. Otherwise, **Destination* is unchanged.

InterlockedCompareExchangePointer provides a fast, atomic way to synchronize the testing and updating of a pointer variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comparand*, the value pointed to by *Destination* is set to the value of *Exchange*.

InterlockedCompareExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchangePointer** is atomic only with respect to other **InterlockedXxx** calls. It does not use a spin lock and can be safely used on pageable data.

The **InterlockedCompareExchangePointer** routine is atomic only with respect to other **InterlockedXxx** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedCompareExchange

InterlockedExchange

InterlockedExchangePointer

InterlockedDecrement function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedDecrement** routine decrements a caller-supplied variable of type LONG as an atomic operation.

Syntax

```
LONG InterlockedDecrement(
  LONG volatile *Addend
);
```

Parameters

`Addend`

A pointer to a variable to be decremented.

Return Value

InterlockedDecrement returns the decremented value.

Remarks

InterlockedDecrement should be used instead of **ExInterlockedDecrementLong** because it is both more efficient and faster.

InterlockedDecrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedDecrement is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedExchange

InterlockedIncrement

InterlockedExchange function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchange** routine sets an integer variable to a given value as an atomic operation.

Syntax

```
LONG InterlockedExchange(
  LONG volatile *Target,
  LONG          Value
);
```

Parameters

`Target`

A pointer to a variable to be set to the supplied *Value* as an atomic operation.

`Value`

Specifies the value to which the variable will be set.

Return Value

InterlockedExchange returns the value of the variable at *Target* when the call occurred.

Remarks

InterlockedExchange should be used instead of **ExInterlockedExchangeUlong**, because it is both faster and more efficient.

InterlockedExchange is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

A call to **InterlockedExchange** routine is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedIncrement

InterlockedExchangeAdd function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchangeAdd** routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.

Syntax

```
LONG InterlockedExchangeAdd(
  LONG volatile *Addend,
  LONG          Value
);
```

Parameters

`Addend`

A pointer to an integer variable.

`Value`

Specifies the value to be added to *Addend*.

Return Value

InterlockedExchangeAdd returns the original value of the *Addend* variable when the call occurred.

Remarks

InterlockedExchangeAdd should be used instead of **ExInterlockedAddUlong** because it is both faster and more efficient.

InterlockedExchangeAdd is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedExchangeAdd is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10

IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedIncrement

InterlockedExchangePointer function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchangePointer** routine performs an atomic operation that sets a pointer to a new value.

Syntax

```
PVOID InterlockedExchangePointer(
  PVOID volatile *Target,
  PVOID          Value
);
```

Parameters

`Target`

A pointer to a PVOID value. The routine sets (**Target*) to *Value*.

`Value`

Specifies the PVOID value to set (**Target*) to.

Return Value

InterlockedExchangePointer returns the original value of the pointer at **Target* (that is, the value of this pointer at entry to the routine).

Remarks

InterlockedExchangePointer provides a fast, atomic way to synchronize updating a pointer variable that is shared by multiple threads.

InterlockedExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedExchangePointer** is atomic only with respect to other **Interlocked*Xxx*** calls. It does not use a spin lock and can be safely used on pageable data.

The *Target* parameter should be aligned on either a 32-bit or 64-bit boundary, depending on the system type, for better performance.

A call to **InterlockedExchangePointer** is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop

Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedCompareExchange

InterlockedCompareExchangePointer

InterlockedExchange

InterlockedIncrement function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedIncrement** routine increments a caller-supplied variable as an atomic operation.

Syntax

```
LONG InterlockedIncrement(
  LONG volatile *Addend
);
```

Parameters

`Addend`

A pointer to a variable of type LONG.

Return Value

InterlockedIncrement returns the incremented value.

Remarks

InterlockedIncrement should be used instead of **ExInterlockedIncrementLong** because it is both more efficient and faster.

InterlockedIncrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedIncrement is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedExchange

InterlockedOr function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedOr** routine atomically computes a bitwise OR operation.

Syntax

```
LONG InterlockedOr(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be ORed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be ORed with the variable that is pointed to by *Destination*.

Return Value

InterlockedOr returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedOr atomically computes ***Destination*|=*Value*.

Interlocked operations cannot be used on non-cached memory.

Requirements

Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedAnd

InterlockedXor

InterlockedXor function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedOr** routine atomically computes a bitwise exclusive OR operation.

Syntax

```
LONG InterlockedXor(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be exclusive ORed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be exclusive ORed with the variable that is pointed to by *Destination*.

Return Value

InterlockedXor returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedXor atomically computes ***Destination*^=*Value***.

Interlocked operations cannot be used on non-cached memory.

Requirements

Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedAnd

InterlockedOr

PINTERFACE_DEREFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceDereference* routine decrements the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_DEREFERENCE PinterfaceDereference;

void PinterfaceDereference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceReference routine to increment the reference count for the interface.

The driver that imports the interface is responsible for calling the *InterfaceDereference* routine to decrement the reference count after the driver is no longer using the interface. For example, a driver that requests a pointer to the interface by sending an IRP_MN_QUERY_INTERFACE request calls *InterfaceDereference*. Also, a driver that receives a pointer to the interface to another driver must call *InterfaceDereference*.

Requirements

Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceReference

PINTERFACE_REFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceReference* routine increments the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_REFERENCE PinterfaceReference;

void PinterfaceReference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceDereference routine to decrement the reference count for the interface.

The driver that exports the interface is responsible for calling *InterfaceReference* to increment the reference count before the driver exports the interface. For example, the driver that initially exported the interface in response to an IRP_MN_QUERY_INTERFACE request calls *InterfaceReference*. Also, a driver that passes a pointer to the interface to another driver must call *InterfaceReference*.

Requirements

Target Platform	Desktop
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

READ_PORT_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_UCHAR** routine reads a number of bytes from the specified port address into a buffer.

Syntax

```
void READ_PORT_BUFFER_UCHAR(
  PUCHAR Port,
  PUCHAR Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of UCHAR values is read.

`Count`

Specifies the number of bytes to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **READ_PORT_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_ULONG** routine reads a number of ULONG values from the specified port address into a buffer.

Syntax

```
void READ_PORT_BUFFER_ULONG(
  PULONG Port,
  PULONG Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of ULONG values is read.

`Count`

Specifies the number of ULONG values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **READ_PORT_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_USHORT** routine reads a number of USHORT values from the specified port address into a buffer.

Syntax

```
void READ_PORT_BUFFER_USHORT(
  PUSHORT Port,
  PUSHORT Buffer,
  ULONG   Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of USHORT values is read.

`Count`

Specifies the number of USHORT values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **READ_PORT_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_UCHAR** routine reads a byte from the specified port address.

Syntax

```
UCHAR READ_PORT_UCHAR(
  PUCHAR Port
);
```

Parameters

`Port`

Return Value

READ_PORT_UCHAR returns the byte that is read from the specified port address.

Remarks

Callers of **READ_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_ULONG** routine reads a ULONG value from the specified port address.

Syntax

```
ULONG READ_PORT_ULONG(
  PULONG Port
);
```

Parameters

`Port`

Return Value

READ_PORT_ULONG returns the ULONG value that is read from the specified port address.

Remarks

Callers of **READ_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_USHORT** routine reads a USHORT value from the specified port address.

Syntax

```
USHORT READ_PORT_USHORT(
  PUSHORT Port
);
```

Parameters

`Port`

Return Value

READ_PORT_USHORT returns the USHORT value that is read from the specified port address.

Remarks

Callers of **READ_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_UCHAR** routine reads a number of bytes from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of UCHAR values is read.

`Count`

Specifies the number of bytes to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **READ_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_ULONG** routine reads a number of ULONG values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of ULONG values is read.

`Count`

Specifies the number of ULONG values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **READ_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_USHORT** routine reads a number of USHORT values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of USHORT values is read.

`Count`

Specifies the number of USHORT values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **READ_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_UCHAR** routine reads a byte from the specified register address.

Syntax

```
UCHAR READ_REGISTER_UCHAR(
  volatile UCHAR *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_UCHAR returns the byte read from the specified register address.

Remarks

Callers of **READ_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_ULONG** routine reads a ULONG value from the specified register address.

Syntax

```
ULONG READ_REGISTER_ULONG(
  volatile ULONG *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_ULONG returns the ULONG value read from the specified register address.

Remarks

Callers of **READ_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_USHORT** routine reads a USHORT value from the specified register address.

Syntax

```
USHORT READ_REGISTER_USHORT(
  volatile USHORT *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_USHORT returns the USHORT value read from the specified register address.

Remarks

Callers of **READ_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_PORT_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_UCHAR** routine writes a number of bytes from a buffer to the specified port.

Syntax

```
void WRITE_PORT_BUFFER_UCHAR(
  PUCHAR Port,
  PUCHAR Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of UCHAR values is to be written.

`Count`

Specifies the number of bytes to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **WRITE_PORT_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_ULONG** routine writes a number of ULONG values from a buffer to the specified port address.

Syntax

```
void WRITE_PORT_BUFFER_ULONG(
  PULONG Port,
  PULONG Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of ULONG values is to be written.

`Count`

Specifies the number of ULONG values to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **WRITE_PORT_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

WRITE_PORT_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_USHORT** routine writes a number of USHORT values from a buffer to the specified port address.

Syntax

```
void WRITE_PORT_BUFFER_USHORT(
  PUSHORT Port,
  PUSHORT Buffer,
  ULONG   Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of USHORT values is to be written.

`Count`

Specifies the number of USHORT values to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **WRITE_PORT_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

WRITE_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_UCHAR** routine writes a byte to the specified port address.

Syntax

```
void WRITE_PORT_UCHAR(
  PUCHAR Port,
  UCHAR  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_ULONG** routine writes a ULONG value to the specified port address.

Syntax

```
void WRITE_PORT_ULONG(
  PULONG Port,
  ULONG  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_USHORT** routine writes a USHORT value to the specified port address.

Syntax

```
void WRITE_PORT_USHORT(
  PUSHORT Port,
  USHORT  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_UCHAR** routine writes a number of bytes from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of UCHAR values is to be written.

`Count`

Specifies the number of bytes to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **WRITE_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_ULONG** routine writes a number of ULONG values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of ULONG values is to be written.

`Count`

Specifies the number of ULONG values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **WRITE_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_USHORT** routine writes a number of USHORT values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of USHORT values is to be written.

`Count`

Specifies the number of USHORT values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **WRITE_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_UCHAR** routine writes a byte to the specified address.

Syntax

```
void WRITE_REGISTER_UCHAR(
  volatile UCHAR *Register,
  UCHAR          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_ULONG** routine writes a ULONG value to the specified address.

Syntax

```
void WRITE_REGISTER_ULONG(
  volatile ULONG *Register,
  ULONG          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_USHORT** routine writes a USHORT value to the specified address.

Syntax

```
void WRITE_REGISTER_USHORT(
  volatile USHORT *Register,
  USHORT          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	miniport.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

ntddk.h header

1/8/2019 • 35 minutes to read • Edit Online

This header is used by kernel. For more information, see:

- Windows kernel ntddk.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
CUSTOM_SYSTEM_EVENT_TRIGGER_INIT	Initializes the **_CUSTOM_SYSTEM_EVENT_TRIGGER_CONFIG** structure.
DbgPrompt	The DbgPrompt routine displays a caller-specified user prompt string on the kernel debugger's display device and obtains a user response string.
ExFreePool	The ExFreePool routine deallocates a block of pool memory.
ExRaiseAccessViolation	The ExRaiseAccessViolation routine can be used with structured exception handling to throw a driver-determined exception for a memory access violation that occurs when a driver processes I/O requests.
ExRaiseDatatypeMisalignment	The ExRaiseDatatypeMisalignment routine can be used with structured exception handling to throw a driver-determined exception for a misaligned data type that occurs when a driver processes I/O requests.
ExUuidCreate	The ExUuidCreate routine initializes a UUID (GUID) structure to a newly generated value.
FsRtlIsTotalDeviceFailure	The FsRtlIsTotalDeviceFailure routine determines whether a media or other hardware failure has occurred.
HalAllocateHardwareCounters	The HalAllocateHardwareCounters routine allocates a set of hardware performance counters.
HalExamineMBR	The HalExamineMBR routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.
HalFreeHardwareCounters	The HalFreeHardwareCounters routine frees a set of hardware performance counters that was acquired in a previous call to HalAllocateHardwareCounters routine.
IoAllocateAdapterChannel	Deprecated. Use PALLOCATE_ADAPTER_CHANNEL.

TITLE	DESCRIPTION
IoAllocateController	The IoAllocateController routine sets up the call to a driver-supplied ControllerControl routine as soon as the device controller, represented by the given controller object, is available to carry out an I/O operation for the target device, represented by the given device object.
IoAssignArcName	The IoAssignArcName routine creates a symbolic link between the ARC name of a physical device and the name of the corresponding device object when it has been created.
IoAttachDeviceToDeviceStackSafe	The IoAttachDeviceToDeviceStackSafe routine attaches the caller's device object to the topmost device object in a driver stack.
IoCancelFileOpen	The IoCancelFileOpen routine can be used by a file system filter driver to close a file that has been opened by a file system driver in the filter driver's device stack.
IoClearActivityIdThread	The IoClearActivityIdThread routine clears the activity ID of the current thread.
IoCreateController	The IoCreateController routine allocates memory for and initializes a controller object with a controller extension of a driver-determined size.
IoCreateDisk	The IoCreateDisk routine initializes a raw disk by creating a new partition table.
IoCreateFileEx	The IoCreateFileEx routine either causes a new file or directory to be created, or opens an existing file, device, directory, or volume and gives the caller a handle for the file object.
IoCreateFileSpecifyDeviceObjectHint	The IoCreateFileSpecifyDeviceObjectHint routine is used by file system filter drivers to send a create request only to the filters below a specified device object and to the file system.
IoDeassignArcName	The IoDeassignArcName routine removes a symbolic link between the ARC name for a device and the named device object.
IoDecrementKeepAliveCount	The IoDecrementKeepAliveCount routine decrements a reference count associated with an Windows app on a specific device.
IoDeleteController	The IoDeleteController routine removes a given controller object from the system, for example, when the driver that created it is being unloaded.
IoFreeController	The IoFreeController routine releases a previously allocated controller object when the driver has completed an I/O request.
IoGetActivityIdIrp	The IoGetActivityIdIrp routine retrieves the current activity ID associated with an IRP.

TITLE	DESCRIPTION
IoGetActivityIdThread	The IoGetActivityIdThread routine returns the activity ID associated with the current thread.
IoGetConfigurationInformation	The IoGetConfigurationInformation routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.
IoGetFileObjectGenericMapping	The IoGetFileObjectGenericMapping routine returns information about the mapping between each generic access right and the set of specific access rights for file objects.
IoGetInitiatorProcess	The IoGetInitiatorProcess routine retrieves the process which initiated the creation of a file object if different than the process which is issuing the create.
IoGetOplockKeyContext	The IoGetOplockKeyContext routine returns a target oplock key context for a file object.
IoGetOplockKeyContextEx	The IoGetOplockKeyContext routine returns a target oplock key context for a file object.
IoGetPagingIoPriority	The IoGetPagingIoPriority routine indicates the priority level of a paging I/O request.
IoGetSiloParameters	This routine indicates if a file is within a Container context.
IoGetTransactionParameterBlock	The IoGetTransactionParameterBlock routine returns the transaction parameter block for a transacted file operation.
IoIncrementKeepAliveCount	The IoIncrementKeepAliveCount routine increments a reference count associated with an Windows app process on a specific device.
IoInitializeDriverCreateContext	The IoInitializeDriverCreateContext routine initializes a caller-allocated variable of type IO_DRIVER_CREATE_CONTEXT.
IoIsFileObjectIgnoringSharing	The IoIsFileObjectIgnoringSharing routine determines if a file object is set with the option to ignore file sharing access checks.
IoIsFileOriginRemote	The IoIsFileOriginRemote routine determines whether a given file object is for a remote create request.
IoIsValidIrpStatus	The IoIsValidIrpStatus routine validates the specified NTSTATUS status code value.
IoMakeAssociatedIrp	This routine is reserved for use by file systems and file system filter drivers.
IoPropagateActivityIdToThread	The IoPropagateActivityIdToThread routine associates the activity ID from an IRP with the current thread.

TITLE	DESCRIPTION
IoQueryFullDriverPath	The IoQueryFullDriverPath routine retrieves the full path name of the binary file that is loaded for the specified driver object.
IoRaiseHardError	The IoRaiseHardError routine causes a dialog box to appears that warns the user that a device I/O error has occurred, which might indicate that a physical device is failing.
IoRaiseInformationalHardError	The IoRaiseInformationalHardError routine sends a dialog box to the user, warning about a device I/O error that indicates why a user I/O request failed.
IoReadDiskSignature	The IoReadDiskSignature routine reads the disk signature information for the partition table of a disk.
IoReadPartitionTable	The IoReadPartitionTable routine is obsolete and is provided only to support existing drivers.
IoReadPartitionTableEx	The IoReadPartitionTableEx routine reads a list of partitions on a disk having a specified sector size and creates an entry in the partition list for each recognized partition.
IoRegisterBootDriverCallback	The IoRegisterBootDriverCallback routine registers a BOOT_DRIVER_CALLBACK_FUNCTION routine to be called during the initialization of a boot-start driver and its dependent DLLs.
IoRegisterBootDriverReinitialization	The IoRegisterBootDriverReinitialization routine is called by a boot driver to register the driver's reinitialization routine with the I/O manager to be called after all devices have been enumerated and started.
IoRegisterDriverReinitialization	The IoRegisterDriverReinitialization routine is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.
IoReportDetectedDevice	The IoReportDetectedDevice routine reports a non-PnP device to the PnP manager.
IoReportResourceForDetection	The IoReportResourceForDetection routine claims hardware resources in the configuration registry for a legacy device.
IoReportRootDevice	The IoReportRootDevice routine reports a device that cannot be detected by a PnP bus driver to the PnP Manager. IoReportRootDevice allows only one device per driver to be created.
IoSetActivityIdIrp	The IoSetActivityIdIrp routine associates an activity ID with an IRP.
IoSetActivityIdThread	The IoSetActivityIdThread routine associates an activity ID with the current thread. Drivers should use this routine when they are tracing aware and are issuing I/O on a worker thread.

TITLE	DESCRIPTION
IoSetFileObjectIgnoreSharing	The IoSetFileObjectIgnoreSharing routine sets a file object to ignore file sharing access checks.
IoSetFileOrigin	The IoSetFileOrigin routine specifies whether a given file object is for a remote create request.
IoSetHardErrorOrVerifyDevice	Lower-level drivers call the IoSetHardErrorOrVerifyDevice routine to identify a removable media device that has encountered an error, so that a file system driver can prompt the user to verify that the medium is valid.
IoSetMasterIrpStatus	The IoSetMasterIrpStatus routine conditionally replaces the Status value in an IRP with the specified NTSTATUS value.
IoSetPartitionInformation	The IoSetPartitionInformation routine is obsolete and is provided only to support existing drivers.
IoSetPartitionInformationEx	For the disk represented by DeviceObject, the IoSetPartitionInformationEx routine initializes a partition table entry with the information specified in the SET_PARTITION_INFORMATION_EX structure.
IoSetSystemPartition	The IoSetSystemPartition routine sets the boot partition for the system.
IoSetThreadHardErrorMode	The IoSetThreadHardErrorMode routine enables or disables hard error reporting for the current thread.
IoTransferActivityId	The IoTransferActivityId routine logs an ETW transfer event using the I/O tracing provider on behalf of the caller. This allows a driver to associate two related activity IDs without requiring a specific provider to be enabled.
IoUnregisterBootDriverCallback	The IoUnRegisterBootDriverCallback routine unregisters a previously registered BOOT_DRIVER_CALLBACK_FUNCTION routine.
IoVerifyPartitionTable	The IoVerifyPartitionTable routine checks the validity of the partition table for a disk.
IoVolumeDeviceToDosName	The IoVolumeDeviceToDosName routine returns the MS-DOS path for a specified device object that represents a file system volume.
IoWritePartitionTable	The IoWritePartitionTable routine is obsolete and is provided only to support existing drivers.
IoWritePartitionTableEx	The IoWritePartitionTableEx routine writes partition tables from the entries in the partition list buffer for each partition on the disk represented by the given device object.
KeAreApcsDisabled	The KeAreApcsDisabled routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.

TITLE	DESCRIPTION
KeBugCheck	The KeBugCheck routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.
KeEnterCriticalRegion	The KeEnterCriticalRegion routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.
KeEnterGuardedRegion	The KeEnterGuardedRegion routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.
KeExpandKernelStackAndCallout	The KeExpandKernelStackAndCallout routine calls a routine with a guaranteed amount of stack space.
KeGetCurrentNodeNumber	The KeGetCurrentNodeNumber routine gets the NUMA node number for the logical processor that the caller is running on.
KeGetCurrentProcessorNumber	The KeGetCurrentProcessorNumber routine returns the system-assigned number of the current processor on which the caller is running.
KeGetCurrentProcessorNumberEx	The KeGetCurrentProcessorNumberEx routine gets the processor number of the logical processor that the caller is running on.
KeInitializeCrashDumpHeader	The KeInitializeCrashDumpHeader routine supplies the header information the system requires for a crash dump file.
KeInvalidateAllCaches	The KeInvalidateAllCaches routine flushes all processor caches.
KeLeaveCriticalRegion	The KeLeaveCriticalRegion routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to KeEnterCriticalRegion.
KeLeaveGuardedRegion	The KeLeaveGuardedRegion routine exits a guarded region entered by KeEnterGuardedRegion.
KePulseEvent	The KePulseEvent routine atomically sets an event object to a signaled state, attempts to satisfy as many waits as possible, and then resets the event object to a not-signaled state.
KeQueryActiveGroupCount	The KeQueryActiveGroupCount routine returns the number of active processor groups in a multiprocessor system.
KeQueryActiveProcessorCount	The KeQueryActiveProcessorCount routine returns the number of currently active processors.
KeQueryActiveProcessorCountEx	The KeQueryActiveProcessorCountEx routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.
KeQueryActiveProcessors	The KeQueryActiveProcessors routine returns a bitmask of the currently active processors.

TITLE	DESCRIPTION
KeQueryGroupAffinity	The KeQueryGroupAffinity routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.
KeQueryHardwareCounterConfiguration	The KeQueryHardwareCounterConfiguration routine queries the operating system for the list of hardware counters to use for thread profiling.
KeQueryHighestNodeNumber	The KeQueryHighestNodeNumber routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryLogicalProcessorRelationship	The KeQueryLogicalProcessorRelationship routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.
KeQueryMaximumGroupCount	The KeQueryMaximumGroupCount routine returns the maximum number of groups in a multiprocessor system.
KeQueryMaximumProcessorCount	The KeQueryMaximumProcessorCount routine returns the maximum number of processors.
KeQueryMaximumProcessorCountEx	The KeQueryMaximumProcessorCountEx routine returns the maximum number of logical processors in a specified group in a multiprocessor system.
KeQueryNodeActiveAffinity	The KeQueryNodeActiveAffinity routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryNodeMaximumProcessorCount	The KeQueryNodeMaximumProcessorCount routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.
KeQueryTickCount	The KeQueryTickCount routine maintains a count of the interval timer interrupts that have occurred since the system was booted.
KeRaiseIrqlToDpcLevel	The KeRaiseIrqlToDpcLevel routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeSetBasePriorityThread	The KeSetBasePriorityThread routine sets the run-time priority, relative to the current process, for a given thread.
KeSetHardwareCounterConfiguration	The KeSetHardwareCounterConfiguration routine specifies a list of hardware counters to use for thread profiling.
KeSetImportanceDpc	The KeSetImportanceDpc routine specifies how soon the DPC routine is run.

TITLE	DESCRIPTION
KeSetTargetProcessorDpc	The KeSetTargetProcessorDpc routine specifies the processor that a DPC routine will be run on.
MmAllocateContiguousMemory	The MmAllocateContiguousMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCache	The MmAllocateContiguousMemorySpecifyCache routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCacheNode	The MmAllocateContiguousMemorySpecifyCacheNode routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousNodeMemory	The MmAllocateContiguousNodeMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateNonCachedMemory	The MmAllocateNonCachedMemory routine allocates a virtual address range of noncached and cache-aligned memory.
MmCopyMemory	The MmCopyMemory routine copies the specified range of virtual or physical memory into the caller-supplied buffer.
MmFreeContiguousMemory	The MmFreeContiguousMemory routine releases a range of physically contiguous memory that was allocated by an MmAllocateContiguousMemoryXxx routine.
MmFreeContiguousMemorySpecifyCache	The MmFreeContiguousMemorySpecifyCache routine frees a buffer that was allocated by an MmAllocateContiguousMemorySpecifyCacheXxx routine.
MmFreeNonCachedMemory	The MmFreeNonCachedMemory routine releases a range of noncached memory that was allocated by the MmAllocateNonCachedMemory routine.
MmGetPhysicalAddress	The MmGetPhysicalAddress routine returns the physical address corresponding to a valid nonpaged virtual address.
MmIsAddressValid	The MmIsAddressValid routine checks whether a page fault will occur for a read or write operation at a given virtual address.Warning We do not recommend using this function.
MmIsThisAnNtAsSystem	The MmIsThisAnNtAsSystem routine is obsolete for Windows XP and later versions of Windows. Use RtlGetVersion or RtlVerifyVersionInfo instead.
MmLockPagableSectionByHandle	The MmLockPagableSectionByHandle routine locks a pageable code or data section into system memory by incrementing the reference count on the handle to the section.
MmMapViewInSystemSpace	The MmMapViewInSystemSpace function maps the specified section into the system's address space.

TITLE	DESCRIPTION
MmSecureVirtualMemory	The MmSecureVirtualMemory routine secures a user-space memory address range so that it cannot be freed and its protection type cannot be made more restrictive.
MmUnmapViewInSystemSpace	
MmUnsecureVirtualMemory	The MmUnsecureVirtualMemory routine unsecures a memory address range secured by the MmSecureVirtualMemory routine.
NtOpenProcess	The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.
PsAllocSiloContextSlot	This routine allocates a slot that can be used to insert, retrieve, and delete an object in all silos. .
PsAttachSiloToCurrentThread	This routine places a thread temporarily into the specified Silo.
PsCreateSiloContext	This routine creates an object that will be inserted in a Silo.
PsDereferenceSiloContext	This routine decrements the reference count on the object.
PsDetachSiloFromCurrentThread	This routine removes a thread from a silo which was added by an attach. For more info about attaching, see the PsAttachSiloToCurrentThread routine.
PsFreeSiloContextSlot	This routine frees the specified slot and makes it available in the system. It undoes the effects of the PsAllocSiloContextSlot routine.
PsGetCurrentProcessId	The PsGetCurrentProcessId routine identifies the current thread's process.
PsGetCurrentServerSilo	This routine returns the effective server silo for the thread.
PsGetCurrentSilo	This routine returns the current silo for the calling thread. First the thread is checked to see if it has been attached to a silo. If not, then the thread is checked to see if it is in a silo.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetCurrentThreadId	The PsGetCurrentThreadId routine identifies the current thread.
PsGetCurrentThreadTeb	The PsGetCurrentThreadTeb routine returns the Thread Environment Block (TEB) of the current thread. The call must be made in kernel-mode.
PsGetEffectiveServerSilo	This routine traverses the parent chain of the Silo until finding the effective server silo or host silo.
PsGetHostSilo	This routine returns the host silo.

TITLE	DESCRIPTION
PsGetJobServerSilo	This routine returns the effective ServerSilo for the job. The returned pointer is valid as long as the supplied Job object remains referenced.
PsGetJobSilo	This routine returns the first job in the hierarchy that is a Silo. The returned pointer is valid as long as the supplied Job object remains referenced.
PsGetParentSilo	Retrieves the most immediate parent silo in the hierarchy for a given job object.
PsGetPermanentSiloContext	This routine retrieves an object that was inserted in the Silo without incrementing the reference count.
PsGetProcessCreateTimeQuadPart	The PsGetProcessCreateTimeQuadPart routine returns a LONGLONG value that represents the time at which the process was created.
PsGetProcessId	The PsGetProcessId routine returns the process identifier (process ID) that is associated with a specified process.
PsGetServerSiloActiveConsoleId	Gets the active console for the current server silo context for the supplied thread.
PsGetSiloContainerId	Gets the ContainerId for the given Silo.
PsGetSiloContext	This routine retrieves the silo context from the specified silo and slot.
PsGetSiloMonitorContextSlot	This routine returns the silo context slot that was allocated by the monitor during the registration.
PsGetThreadProcessId	The PsGetThreadProcessId routine returns the process identifier that is associated with a specified thread's process.
PsGetThreadServerSilo	Determines whether the given thread is a silo.
PshedAllocateMemory	The PshedAllocateMemory function allocates a block of memory from the nonpaged pool.
PshedFreeMemory	The PshedFreeMemory function frees a block of memory that was previously allocated by calling the PshedAllocateMemory function.
PshedIsSystemWheaEnabled	The PshedIsSystemWheaEnabled function returns a Boolean value that indicates whether the system is WHEA-enabled.
PshedRegisterPlugin	The PshedRegisterPlugin function registers a PSHED plug-in with the PSHED.
PshedSynchronizeExecution	The PshedSynchronizeExecution function synchronizes the execution of a given function with the hardware error processing for an error source.

TITLE	DESCRIPTION
PsInsertPermanentSiloContext	This routine inserts an object in an empty slot in a Silo.
PsInsertSiloContext	This routine inserts an object in an empty slot in a Silo.
PsIsHostSilo	This routine will check if the supplied Silo is the host silo.
PsMakeSiloContextPermanent	This routine makes the slot in a silo instance read-only, allowing the object in the slot to be retrieved without affecting the reference count on that object.
PsReferenceSiloContext	This routine increments the reference count on the object.
PsRegisterSiloMonitor	This routine registers a server silo monitor that can receive notifications about server silo events.
PsRemoveCreateThreadNotifyRoutine	The PsRemoveCreateThreadNotifyRoutine routine removes a callback routine that was registered by the PsSetCreateThreadNotifyRoutine routine.
PsRemoveLoadImageNotifyRoutine	The PsRemoveLoadImageNotifyRoutine routine removes a callback routine that was registered by the PsSetLoadImageNotifyRoutine routine.
PsRemoveSiloContext	This routine removes an object that was inserted in the Silo.
PsReplaceSiloContext	This routine inserts an object in a Silo.
PsSetCreateProcessNotifyRoutine	The PsSetCreateProcessNotifyRoutine routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.
PsSetCreateProcessNotifyRoutineEx	The PsSetCreateProcessNotifyRoutineEx routine registers or removes a callback routine that notifies the caller when a process is created or exits.
PsSetCreateProcessNotifyRoutineEx2	The PsSetCreateProcessNotifyRoutineEx2 routine registers or removes a callback routine that notifies the caller when a process is created or deleted.
PsSetCreateThreadNotifyRoutine	The PsSetCreateThreadNotifyRoutine routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.
PsSetCreateThreadNotifyRoutineEx	The PsSetCreateThreadNotifyRoutineEx routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.
PsSetLoadImageNotifyRoutine	The PsSetLoadImageNotifyRoutine routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
PsSetLoadImageNotifyRoutineEx	The PsSetLoadImageNotifyRoutineEx routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).

TITLE	DESCRIPTION
PsStartSiloMonitor	This routine tries to start the server silo monitor.
PsTerminateServerSilo	This routine terminates the specified silo.
PsUnregisterSiloMonitor	This routine unregisters a server silo monitor.
RtlCharToInteger	The RtlCharToInteger routine converts a single-byte character string to an integer value in the specified base.
RtlCompareString	The RtlCompareString routine compares two counted strings.
RtlConvertLongToLuid	The RtlConvertLongToLuid routine converts a long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.
RtlConvertUlongToLuid	The RtlConvertUlongToLuid routine converts an unsigned long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.
RtlCopyString	The RtlCopyString routine copies a source string to a destination string.
RtlDelete	The RtlDelete routine deletes the specified node from the splay link tree.
RtlDeleteElementGenericTable	The RtlDeleteElementGenericTable routine deletes an element from a generic table.
RtlDeleteElementGenericTableAvl	The RtlDeleteElementGenericTableAvl routine deletes an element from a generic table.
RtlDeleteNoSplay	The RtlDeleteNoSplay routine deletes the specified node from the splay link tree.
RtlDrainNonVolatileFlush	The routine RtlDrainNonVolatileFlush waits for the flushes initiated by RtlFlushNonVolatileMemory to finish.
RtlEnumerateGenericTable	The RtlEnumerateGenericTable routine is used to enumerate the elements in a generic table.
RtlEnumerateGenericTableAvl	The RtlEnumerateGenericTableAvl routine is used to enumerate the elements in a generic table.
RtlEnumerateGenericTableLikeADirectory	The RtlEnumerateGenericTableLikeADirectory routine returns the elements of a generic table, one-by-one, in collation order.
RtlEnumerateGenericTableWithoutSplaying	The RtlEnumerateGenericTableWithoutSplaying routine is used to enumerate the elements in a generic table.
RtlEnumerateGenericTableWithoutSplayingAvl	The RtlEnumerateGenericTableWithoutSplayingAvl routine is used to enumerate the elements in a generic table.

TITLE	DESCRIPTION
RtlEqualString	The RtlEqualString routine compares two counted strings to determine whether they are equal.
RtlExtendCorrelationVector	This routine extends the supplied correlation vector. For a correlation vector of the form X.i, the extended value is X.i.0.
RtlFlushNonVolatileMemory	The routine RtlFlushNonVolatileMemory optimally flushes the given non-volatile memory region.
RtlFlushNonVolatileMemoryRanges	The routine RtlFlushNonVolatileMemoryRanges optimally flushes the given non-volatile memory regions.
RtlFreeNonVolatileToken	The routine RtlFreeNonVolatileToken is a cleanup function for the opaque NvToken which is given by a successful call to RtlGetNonVolatileToken.
RtlGetElementGenericTable	The RtlGetElementGenericTable routine returns a pointer to the caller-supplied data for a particular generic table element.
RtlGetElementGenericTableAvl	The RtlGetElementGenericTableAvl routine returns a pointer to the caller-supplied data for a particular generic Adelson-Velsky/Landis (AVL) table element.
RtlGetEnabledExtendedFeatures	The RtlGetEnabledExtendedFeatures routine returns a mask of extended processor features that are enabled by the system.
RtlGetNonVolatileToken	The routine, RtlGetNonVolatileToken, gets various properties about a non-volatile memory buffer and stores them in the variable NvToken.
RtlGetPersistedStateLocation	Retrieves the redirected location for the specified SourceID from the redirection map.
RtlIncrementCorrelationVector	Increments the specified correlation vector. For a correlation vector of the form X.i, the incremented value is be X.(i+1).
RtlInitializeCorrelationVector	Initializes the specified correlation vector with the supplied GUID.
RtlInitializeGenericTable	The RtlInitializeGenericTable routine initializes a generic table.
RtlInitializeGenericTableAvl	The RtlInitializeGenericTableAvl routine initializes a generic table using Adelson-Velsky/Landis (AVL) trees.
RtlInitializeSplayLinks	The RtlInitializeSplayLinks routine initializes a splay link node.
RtlInsertAsLeftChild	The RtlInsertAsLeftChild routine inserts a splay link node into the tree as the left child of the specified node.
RtlInsertAsRightChild	The RtlInsertAsRightChild routine inserts a given splay link into the tree as the right child of a given node in that tree.
RtlInsertElementGenericTable	The RtlInsertElementGenericTable routine adds a new element to a generic table.

TITLE	DESCRIPTION
RtlInsertElementGenericTableAvl	The RtlInsertElementGenericTableAvl routine adds a new entry to a generic table.
RtlInsertElementGenericTableFullAvl	The RtlInsertElementGenericTableFullAvl routine adds a new entry to a generic table.
RtlIsGenericTableEmpty	The RtlIsGenericTableEmpty routine determines if a generic table is empty.
RtlIsGenericTableEmptyAvl	The RtlIsGenericTableEmptyAvl routine determines if a generic table is empty.
RtlIsLeftChild	The RtlIsLeftChild routine determines whether a given splay link is the left child of a node in a splay link tree.
RtlIsRightChild	The RtlIsRightChild routine determines whether a given splay link is the right child of a node in a splay link tree.
RtlIsRoot	The RtlIsRoot routine determines whether the specified node is the root node of a splay link tree.
RtlIsStateSeparationEnabled	Checks if the SKU for the current context supports multiple sessions.
RtlLeftChild	The RtlLeftChild routine returns a pointer to the left child of the specified splay link node.
RtlLookupElementGenericTable	The RtlLookupElementGenericTable routine searches a generic table for an element that matches the specified data.
RtlLookupElementGenericTableAvl	The RtlLookupElementGenericTableAvl routine searches a generic table for an element that matches the specified data.
RtlLookupElementGenericTableFullAvl	The RtlLookupElementGenericTableFullAvl routine searches a generic table for an element that matches the specified data.
RtlLookupFirstMatchingElementGenericTableAvl	The RtlLookupFirstMatchingElementGenericTableAvl routine finds the left-most element in the tree that matches the indicated data.
RtlMapGenericMask	The RtlMapGenericMask routine determines the nongeneric access rights specified by an ACCESS_MASK.
RtlNumberGenericTableElements	The RtlNumberGenericTableElements routine returns the number of elements in a generic table.
RtlNumberGenericTableElementsAvl	The RtlNumberGenericTableElementsAvl routine returns the number of elements in a generic table.
RtlParent	The RtlParent routine returns a pointer to the parent of the specified node in a splay link tree.

TITLE	DESCRIPTION
RtlPrefixUnicodeString	The RtlPrefixUnicodeString routine compares two Unicode strings to determine whether one string is a prefix of the other.
RtlQueryRegistryValueWithFallback	Retrieves a value entry for a registry key by using, a primary handle; if not found, uses the fallback handle.
RtlRaiseCustomSystemEventTrigger	Allows NT services and kernel-mode and user-mode drivers to raise a custom trigger for a device.
RtlRealPredecessor	The RtlRealPredecessor routine returns a pointer to the predecessor of the specified node in the splay link tree.
RtlRealSuccessor	The RtlRealSuccessor routine returns a pointer to the successor of the specified node in the splay link tree.
RtlRightChild	The RtlRightChild routine returns a pointer to the right child of the specified splay link node.
RtlRunOnceBeginInitialize	The RtlRunOnceBeginInitialize routine begins a one-time initialization.
RtlRunOnceComplete	The RtlRunOnceComplete routine completes the one-time initialization began by RtlRunOnceBeginInitialize.
RtlRunOnceExecuteOnce	The RtlRunOnceExecuteOnce performs a one-time initialization.
RtlRunOnceInitialize	The RtlRunOnceInitialize routine initializes a RTL_RUN_ONCE structure.
RtlSplay	The RtlSplay routine rebalances, or "splays," a splay link tree around the specified splay link, making that link the new root of the tree.
RtlSubtreePredecessor	The RtlSubtreePredecessor routine returns a pointer to the predecessor of the specified node within the subtree that is rooted at that node.
RtlSubtreeSuccessor	The RtlSubtreeSuccessor routine returns a pointer to the successor of the specified node within the subtree that is rooted at that node.
RtlUpcaseUnicodeString	The RtlUpcaseUnicodeString routine converts a copy of the source string to uppercase and writes the converted string in the destination buffer.
RtlUpperChar	The RtlUpperChar routine converts the specified character to uppercase.
RtlUpperString	The RtlUpperString routine copies the given SourceString to the DestinationString buffer, converting it to uppercase.

TITLE	DESCRIPTION
RtlValidateCorrelationVector	Validates the specified correlation vector to check whether it conforms to the Correlation Vector Specification (v2).
RtlVolumeDeviceToDosName	The RtlVolumeDeviceToDosName routine is obsolete for Windows XP and later versions of Windows. Use IoVolumeDeviceToDosName instead.RtlVolumeDeviceToDosName returns the MS-DOS path for a specified device object that represents a file system volume.
RtlWriteNonVolatileMemory	The routine RtlWriteNonVolatileMemory copies the contents of a source buffer to a non-volatile destination memory buffer.
SeSinglePrivilegeCheck	The SeSinglePrivilegeCheck routine checks for the passed privilege value in the context of the current thread.
WheaFindErrorRecordSection	The WheaFindErrorRecordSection function searches for a specified Windows Hardware Error Architecture (WHEA) error record section within a WHEA error record. The error record section is formatted as a WHEA_ERROR_RECORD_SECTION_DESCRIPTOR structure.
WheaFindNextErrorRecordSection	The WheaFindNextErrorRecordSection function allows a caller to iteratively examine the WHEA error record sections within a WHEA error record. Each error record section is formatted as a WHEA_ERROR_RECORD_SECTION_DESCRIPTOR structure.
WheaGetErrPacketFromErrRecord	The WheaGetErrPacketFromErrRecord function returns a pointer to the hardware error packet that is contained within a WHEA error record. The hardware error packet is formatted as a WHEA_ERROR_PACKET structure.
WheaIsValidErrorRecordSignature	The WheaIsValidErrorRecordSignature function verifies whether a WHEA error record is valid.
ZwAllocateLocallyUniqueId	The ZwAllocateLocallyUniqueId routine allocates a locally unique identifier (LUID).
ZwDeviceIoControlFile	The ZwDeviceIoControlFile routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.
ZwOpenProcess	The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.
ZwPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.
ZwQueryVolumeInformationFile	The ZwQueryVolumeInformationFile routine retrieves information about the volume associated with a given file, directory, storage device, or volume.
ZwSetInformationThread	The ZwSetInformationThread routine sets the priority of a thread.

TITLE	DESCRIPTION
ZwTerminateProcess	The ZwTerminateProcess routine terminates a process and all of its threads.

Callback functions

TITLE	DESCRIPTION
BOOT_DRIVER_CALLBACK_FUNCTION	
DRIVER_REINITIALIZE	The Reinitialize routine continues driver and device initialization after the driver's DriverEntry routine returns.
EXPAND_STACK_CALLOUT	The ExpandedStackCall routine executes with a guaranteed stack size.
PCREATE_PROCESS_NOTIFY_ROUTINE	Process-creation callback implemented by a driver to track the system-wide creation and deletion of processes against the driver's internal state.
PCREATE_PROCESS_NOTIFY_ROUTINE_EX	A callback routine implemented by a driver to notify the caller when a process is created or exits.
PCREATE_THREAD_NOTIFY_ROUTINE	A callback routine implemented by a driver to notify the caller when a thread is created or deleted.
PGET_LOCATION_STRING	The PnpGetLocationString routine provides the device-specific part of the device's SPDRP_LOCATION_PATHS property.
pHalExamineMBR	The HalExamineMBR routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.
pHalQuerySystemInformation	
pHalSetSystemInformation	Register MCA driver with the HAL.
PLOAD_IMAGE_NOTIFY_ROUTINE	Called by the operating system to notify the driver when a driver image or a user image (for example, a DLL or EXE) is mapped into virtual memory.
PSHED_PI_ATTEMPT_ERROR_RECOVERY	A PSHED plug-in's AttemptRecovery callback function attempts to recover from a recoverable hardware error.
PSHED_PI_CLEAR_ERROR_RECORD	A PSHED plug-in's ClearErrorRecord callback function clears the specified error record from the system's persistent data storage.
PSHED_PI_CLEAR_ERROR_STATUS	A PSHED plug-in's ClearErrorStatus callback function clears any platform-specific error status registers for a corrected hardware error condition.
PSHED_PI_DISABLE_ERROR_SOURCE	A PSHED plug-in's DisableErrorSource callback function disables an error source.

TITLE	DESCRIPTION
PSHED_PI_ENABLE_ERROR_SOURCE	A PSHED plug-in's EnableErrorSource callback function enables an error source.
PSHED_PI_FINALIZE_ERROR_RECORD	A PSHED plug-in's FinalizeErrorRecord callback function adds supplementary error record sections to an error record that more fully describe the error condition.
PSHED_PI_GET_ALL_ERROR_SOURCES	A PSHED plug-in's GetAllErrorSources callback function returns a list of error source descriptor structures that represents all of the error sources that are implemented by the hardware platform.
PSHED_PI_GET_ERROR_SOURCE_INFO	A PSHED plug-in's GetErrorSourceInfo callback function returns an error source descriptor structure that represents a particular error source that is implemented by the hardware platform.
PSHED_PI_GET_INJECTION_CAPABILITIES	A PSHED plug-in's GetInjectionCapabilities callback function returns an error injection capabilities union that describes the types of hardware errors that can be injected into the hardware platform.
PSHED_PI_INJECT_ERROR	A PSHED plug-in's InjectError callback function injects an error into the hardware platform.
PSHED_PI_READ_ERROR_RECORD	A PSHED plug-in's ReadErrorRecord callback function reads an error record from the system's persistent data storage.
PSHED_PI_RETRIEVE_ERROR_INFO	A PSHED plug-in's RetrieveErrorInfo callback function retrieves platform-specific error information about a hardware error that has occurred.
PSHED_PI_SET_ERROR_SOURCE_INFO	A PSHED plug-in's SetErrorSourceInfo callback function configures an error source.
PSHED_PI_WRITE_ERROR_RECORD	A PSHED plug-in's WriteErrorRecord callback function writes an error record to the system's persistent data storage.
RTL_RUN_ONCE_INIT_FN	The RunOnceInitialization routine performs a one-time initialization operation.
SILO_CONTEXT_CLEANUP_CALLBACK	This callback is invoked when the context object reaches a reference count of zero.
SILO_MONITOR_CREATE_CALLBACK	This is callback is invoked when a new silo is created.
SILO_MONITOR_TERMINATE_CALLBACK	This callback is invoked when a silo is terminated.

Structures

TITLE	DESCRIPTION

TITLE	DESCRIPTION
WHEA_ERROR_RECORD	The WHEA_ERROR_RECORD structure describes an error record that contains error information about a hardware error condition that occurred.
WHEA_ERROR_SOURCE_DESCRIPTOR	The WHEA_ERROR_SOURCE_DESCRIPTOR structure describes an error source.
BDCB_IMAGE_INFORMATION	The BDCB_IMAGE_INFORMATION structure describes information about a boot-start driver that is about to be initialized, provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.
BDCB_STATUS_UPDATE_CONTEXT	The BDCB_STATUS_UPDATE_CONTEXT structure describes a status update provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.
CONTROLLER_OBJECT	A controller object represents a hardware adapter or controller with homogenous devices that are the actual targets for I/O requests.
CORRELATION_VECTOR	Store the correlation vector that is used to reference events and the generated logs for diagnostic purposes.
CUSTOM_SYSTEM_EVENT_TRIGGER_CONFIG	Contains information about a custom system event trigger.
DEBUG_TRANSPORT_DATA	Describes debug-transport specific data for use by the transport.
DISK_SIGNATURE	DISK_SIGNATURE contains the disk signature information for a disk's partition table.
FILE_ALIGNMENT_INFORMATION	The FILE_ALIGNMENT_INFORMATION structure is used as an argument to the ZwQueryInformationFile routine.
FILE_ATTRIBUTE_TAG_INFORMATION	The FILE_ATTRIBUTE_TAG_INFORMATION structure is used as an argument to ZwQueryInformationFile.
FILE_DISPOSITION_INFORMATION	The FILE_DISPOSITION_INFORMATION structure is used as an argument to the ZwSetInformationFile routine.
FILE_DISPOSITION_INFORMATION_EX	The FILE_DISPOSITION_INFORMATION_EX structure is used as an argument to the ZwSetInformationFileEx routine and indicates how the operating system should delete a file.
FILE_END_OF_FILE_INFORMATION	The FILE_END_OF_FILE_INFORMATION structure is used as an argument to the ZwSetInformationFile routine.
FILE_FS_FULL_SIZE_INFORMATION	The FILE_FS_FULL_SIZE_INFORMATION structure is used to query sector size information for a file system volume.
FILE_FS_FULL_SIZE_INFORMATION_EX	Used in a request to query sector size information for a file system volume.

TITLE	DESCRIPTION
FILE_FS_LABEL_INFORMATION	The FILE_FS_LABEL_INFORMATION structure is used to set the label for a file system volume.
FILE_FS_OBJECTID_INFORMATION	The FILE_FS_OBJECTID_INFORMATION structure is used to query or set the object ID for a file system volume.
FILE_FS_SECTOR_SIZE_INFORMATION	The FILE_FS_SECTOR_SIZE_INFORMATION structure is used to query physical and logical sector size information for a file system volume.
FILE_FS_SIZE_INFORMATION	The FILE_FS_SIZE_INFORMATION structure is used to query sector size information for a file system volume.
FILE_FS_VOLUME_INFORMATION	The FILE_FS_VOLUME_INFORMATION structure is used to query information about a volume on which a file system is mounted.
FILE_NAME_INFORMATION	The FILE_NAME_INFORMATION structure is used as argument to the ZwQueryInformationFile and ZwSetInformationFile routines.
FILE_VALID_DATA_LENGTH_INFORMATION	The FILE_VALID_DATA_LENGTH_INFORMATION structure is used as an argument to ZwSetInformationFile.
HARDWARE_COUNTER	The HARDWARE_COUNTER structure contains information about a hardware counter.
IMAGE_INFO	Used by driver's load-image routine (PLOAD_IMAGE_NOTIFY_ROUTINE) to specify image information.
IMAGE_INFO_EX	Extended version of the image information structure (see IMAGE_INFO).
IO_DRIVER_CREATE_CONTEXT	The IO_DRIVER_CREATE_CONTEXT structure is used to pass additional parameters to the IoCreateFileEx and FltCreateFileEx2 routines.
IO_FOEXT_SILO_PARAMETERS	This structure describes the Container context that's identified by the IoGetSiloParameters routine.
KEY_CACHED_INFORMATION	The KEY_CACHED_INFORMATION structure holds the cached information available for a registry key or subkey.
KEY_NAME_INFORMATION	The KEY_NAME_INFORMATION structure holds the name and name length of the key.
KEY_VIRTUALIZATION_INFORMATION	The KEY_VIRTUALIZATION_INFORMATION structure defines the basic information that is available for a registry key or subkey.
MM_COPY_ADDRESS	The MM_COPY_ADDRESS structure contains either a virtual memory address or a physical memory address.

TITLE	DESCRIPTION
OPLOCK_KEY_CONTEXT	The OPLOCK_KEY_CONTEXT structure is returned from IoGetOplockKeyContextEx. This structure contains oplock keys for a specific file object.
PCI_EXPRESS_CAPABILITIES_REGISTER	The PCI_EXPRESS_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) capabilities register of a PCIe capability structure.
PCI_EXPRESS_CAPABILITY	The PCI_EXPRESS_CAPABILITY structure describes a PCI Express (PCIe) capability structure.
PCI_EXPRESS_DESIGNATED_VENDOR_SPECIFIC_CAPABILITY	Represents the Designated Vendor-Specific Extended Capability defined by PCI-SIG.
PCI_EXPRESS_DESIGNATED_VENDOR_SPECIFIC_HEADER_1	Represents the Designated Vendor-Specific Extended Capability Header 1 defined by PCI-SIG.
PCI_EXPRESS_DESIGNATED_VENDOR_SPECIFIC_HEADER_2	Represents the Designated Vendor-Specific Extended Capability Header 2 defined by PCI-SIG.
PCI_EXPRESS_DEVICE_CAPABILITIES_REGISTER	The PCI_EXPRESS_DEVICE_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) device capabilities register of a PCIe capability structure.
PCI_EXPRESS_DEVICE_CONTROL_REGISTER	The PCI_EXPRESS_DEVICE_CONTROL_REGISTER structure describes a PCI Express (PCIe) device control register of a PCIe capability structure.
PCI_EXPRESS_DEVICE_STATUS_REGISTER	The PCI_EXPRESS_DEVICE_STATUS_REGISTER structure describes a PCI Express (PCIe) device status register of a PCIe capability structure.
PCI_EXPRESS_LINK_CAPABILITIES_REGISTER	The PCI_EXPRESS_LINK_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) link capabilities register of a PCIe capability structure.
PCI_EXPRESS_LINK_CONTROL_REGISTER	The PCI_EXPRESS_LINK_CONTROL_REGISTER structure describes a PCI Express (PCIe) link control register of a PCIe capability structure.
PCI_EXPRESS_LINK_STATUS_REGISTER	The PCI_EXPRESS_LINK_STATUS_REGISTER structure describes a PCI Express (PCIe) link status register of a PCIe capability structure.
PCI_EXPRESS_PME_REQUESTOR_ID	The PCI_EXPRESS_PME_REQUESTOR_ID structure describes the identifier of the requester of a power management event (PME).
PCI_EXPRESS_ROOT_CAPABILITIES_REGISTER	The PCI_EXPRESS_ROOT_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) root capabilities register of a PCIe capability structure.
PCI_EXPRESS_ROOT_CONTROL_REGISTER	The PCI_EXPRESS_ROOT_CONTROL_REGISTER structure describes a PCI Express (PCIe) root control register of a PCIe capability structure.

TITLE	DESCRIPTION
PCI_EXPRESS_ROOT_STATUS_REGISTER	The PCI_EXPRESS_ROOT_STATUS_REGISTER structure describes a PCI Express (PCIe) root status register of a PCIe capability structure.
PCI_EXPRESS_SLOT_CAPABILITIES_REGISTER	The PCI_EXPRESS_SLOT_CAPABILITIES_REGISTER structure describes a PCI Express (PCIe) slot capabilities register of a PCIe capability structure.
PCI_EXPRESS_SLOT_CONTROL_REGISTER	The PCI_EXPRESS_SLOT_CONTROL_REGISTER structure describes a PCI Express (PCIe) slot control register of a PCIe capability structure.
PCI_EXPRESS_SLOT_STATUS_REGISTER	The PCI_EXPRESS_SLOT_STATUS_REGISTER structure describes a PCI Express (PCIe) slot status register of a PCIe capability structure.
PCI_FPB_CAPABILITIES_REGISTER	The Flattening Portal Bridge (FPB) Capabilities register. See section 7.y.2.
PCI_FPB_CAPABILITY	Flattening Portal Bridge (FPB) Capabilities that is required for any bridge Function that implements FPB. See section 7.y.
PCI_FPB_CAPABILITY_HEADER	The Flattening Portal Bridge (FPB) Capabilities header. See section 7.y.1.
PCI_FPB_MEM_HIGH_VECTOR_CONTROL1_REGISTER	The FPB MEM High Vector Control 1 Register. See section 7.y.6.
PCI_FPB_MEM_HIGH_VECTOR_CONTROL2_REGISTER	The FPB MEM High Vector Control 2 Register. See section 7.y.7.
PCI_FPB_MEM_LOW_VECTOR_CONTROL_REGISTER	FPB MEM Low Vector Control Register. See section 7.y.5.
PCI_FPB_RID_VECTOR_CONTROL1_REGISTER	The FPB RID Vector Control 1 Register. See section 7.y.3.
PCI_FPB_RID_VECTOR_CONTROL2_REGISTER	The FPB RID Vector Control 1 Register. See section 7.y.3.
PCI_FPB_VECTOR_ACCESS_CONTROL_REGISTER	The FPB Vector Access Control Register. See section 7.y.8.
PCI_FPB_VECTOR_ACCESS_DATA_REGISTER	The FPB Vector Access Data Register. See section 7.y.9.
PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR	The PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure describes the counter resources available on the platform.
PHYSICAL_COUNTER_RESOURCE_LIST	The PHYSICAL_COUNTER_RESOURCE_LIST structure describes an array of PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structures.
PNP_LOCATION_INTERFACE	The PNP_LOCATION_INTERFACE structure describes the GUID_PNP_LOCATION_INTERFACE interface.
POWER_THROTTLING_PROCESS_STATE	Stores the throttling policies and how to apply them to a target process when that process is subject to power management.

TITLE	DESCRIPTION
POWER_THROTTLING_THREAD_STATE	Stores the throttling policies and how to apply them to a target thread when that thread is subject to power management.
PROCESS_MITIGATION_CHILD_PROCESS_POLICY	Stores policy information about creating child processes.
PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY	Stores information about process mitigation policy.
PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY	This structure is not supported.
PS_CREATE_NOTIFY_INFO	The PS_CREATE_NOTIFY_INFO structure provides information about a newly created process.
RTL_AVL_TABLE	The RTL_AVL_TABLE structure contains file system-specific data for an Adelson-Velsky/Landis (AVL) tree.
RTL_GENERIC_TABLE	The RTL_GENERIC_TABLE structure contains file system-specific data for a splay tree.
RTL_SPLAY_LINKS	The RTL_SPLAY_LINKS structure is an opaque structure and is used by the system to represent a splay link tree node.
SILO_MONITOR_REGISTRATION	This structure specifies a server silo monitor that can receive notifications about server silo events.
SOC_SUBSYSTEM_FAILURE_DETAILS	The SOC_SUBSYSTEM_FAILURE_DETAILS structure holds information related to a System on a Chip (SoC) bug code.
TXN_PARAMETER_BLOCK	The TXN_PARAMETER_BLOCK structure contains information about a transacted file operation.
WHEA_AER_BRIDGE_DESCRIPTOR	The WHEA_AER_BRIDGE_DESCRIPTOR structure describes a PCI Express (PCIe) bridge error source.
WHEA_AER_ENDPOINT_DESCRIPTOR	The WHEA_AER_ENDPOINT_DESCRIPTOR structure describes a PCI Express (PCIe) endpoint error source.
WHEA_AER_ROOTPORT_DESCRIPTOR	The WHEA_AER_ROOTPORT_DESCRIPTOR structure describes a PCI Express (PCIe) root port error source.
WHEA_ERROR_INJECTION_CAPABILITIES	The WHEA_ERROR_INJECTION_CAPABILITIES union describes the types of hardware errors that can be injected into a hardware platform.
WHEA_ERROR_PACKET_FLAGS	The WHEA_ERROR_PACKET_FLAGS union defines the error condition reported through a WHEA_ERROR_PACKET structure.
WHEA_ERROR_PACKET_V1	The WHEA_ERROR_PACKET_V1 structure describes the hardware error data that is passed to the operating system by a low-level hardware error handler (LLHEH).Note The WHEA_ERROR_PACKET_V1 structure is supported in Windows Server 2008 and Windows Vista SP1.

TITLE	DESCRIPTION
WHEA_ERROR_PACKET_V2	The WHEA_ERROR_PACKET_V1 structure describes the hardware error data that is passed to the operating system by a low-level hardware error handler (LLHEH).Note The WHEA_ERROR_PACKET_V1 structure is supported in Windows Server 2008 and Windows Vista SP1.
WHEA_ERROR_RECORD_HEADER	The WHEA_ERROR_RECORD_HEADER structure describes general information about a hardware error condition.
WHEA_ERROR_RECORD_HEADER_VALIDBITS	The WHEA_ERROR_RECORD_HEADER_VALIDBITS union describes which members of a WHEA_ERROR_RECORD_HEADER structure contain valid data.
WHEA_ERROR_RECORD_SECTION_DESCRIPTOR	The WHEA_ERROR_RECORD_SECTION_DESCRIPTOR structure describes a section of error information that is part of an error record.
WHEA_ERROR_RECORD_SECTION_DESCRIPTOR_VALIDBITS	The WHEA_ERROR_RECORD_SECTION_DESCRIPTOR_VALIDBITS union describes which members of a WHEA_ERROR_RECORD_SECTION_DESCRIPTOR structure contain valid data.
WHEA_ERROR_STATUS	The WHEA_ERROR_STATUS union describes generic error codes that are abstracted from the data contained in implementation-specific error registers.
WHEA_FIRMWARE_ERROR_RECORD_REFERENCE	The WHEA_FIRMWARE_ERROR_RECORD_REFERENCE structure describes a reference to a firmware error record that is specific to the Itanium processor architecture.
WHEA_GENERIC_ERROR	The WHEA_GENERIC_ERROR structure describes error status data for a generic error source.
WHEA_GENERIC_ERROR_BLOCKSTATUS	The WHEA_GENERIC_ERROR_BLOCKSTATUS union indicates what kind of error data is reported in a generic error status block.
WHEA_GENERIC_ERROR_DATA_ENTRY_V2	The WHEA_GENERIC_ERROR_DATA_ENTRY structure describes an error data section in a generic error status block.
WHEA_GENERIC_ERROR_DESCRIPTOR	The WHEA_GENERIC_ERROR_DESCRIPTOR structure describes a generic error source.
WHEA_IPF_CMC_DESCRIPTOR	The WHEA_IPF_CMC_DESCRIPTOR structure describes a corrected machine check (CMC) error source for an Itanium processor.
WHEA_IPF_CPE_DESCRIPTOR	The WHEA_IPF_CPE_DESCRIPTOR structure describes a corrected platform error (CPE) error source for an Itanium processor.
WHEA_IPF_MCA_DESCRIPTOR	The WHEA_IPF_MCA_DESCRIPTOR structure describes a machine check abort (MCA) error source for an Itanium processor.

TITLE	DESCRIPTION
WHEA_MEMORY_ERROR_SECTION	The WHEA_MEMORY_ERROR_SECTION structure describes platform memory error data.
WHEA_MEMORY_ERROR_SECTION_VALIDBITS	The WHEA_MEMORY_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_MEMORY_ERROR_SECTION structure contain valid data.
WHEA_NMI_ERROR_SECTION	The WHEA_NMI_ERROR_SECTION structure describes nonmaskable interrupt (NMI) error data.
WHEA_NOTIFICATION_DESCRIPTOR	The WHEA_NOTIFICATION_DESCRIPTOR structure describes the notification mechanism that is used by an error source.
WHEA_PCI_SLOT_NUMBER	The WHEA_PCI_SLOT_NUMBER structure describes a logical PCI slot.
WHEA_PCIEXPRESS_ERROR_SECTION	The WHEA_PCIEXPRESS_ERROR_SECTION structure describes PCI Express (PCIe) error data.
WHEA_PCIEXPRESS_ERROR_SECTION_VALIDBITS	The WHEA_PCIEXPRESS_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_PCIEXPRESS_ERROR_SECTION structure contain valid data.
WHEA_PCIXBUS_ERROR_SECTION	The WHEA_PCIXBUS_ERROR_SECTION structure describes PCI or PCI-X bus error data.
WHEA_PCIXBUS_ERROR_SECTION_VALIDBITS	The WHEA_PCIXBUS_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_PCIXBUS_ERROR_SECTION structure contain valid data.
WHEA_PCIXDEVICE_ERROR_SECTION	The WHEA_PCIXDEVICE_ERROR_SECTION structure describes PCI or PCI-X device error data.
WHEA_PCIXDEVICE_ERROR_SECTION_VALIDBITS	The WHEA_PCIXDEVICE_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_PCIXDEVICE_ERROR_SECTION structure contain valid data.
WHEA_PERSISTENCE_INFO	The WHEA_PERSISTENCE_INFO union describes data that is used by the error record persistence interface for storing an error record.
WHEA_PROCESSOR_FAMILY_INFO	The WHEA_PROCESSOR_FAMILY_INFO union describes the processor family information for an x86 or x64 processor.
WHEA_PROCESSOR_GENERIC_ERROR_SECTION	Describes processor error data that is not specific to a particular processor architecture.
WHEA_PROCESSOR_GENERIC_ERROR_SECTION_VALIDBITS	The WHEA_PROCESSOR_GENERIC_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_PROCESSOR_GENERIC_ERROR_SECTION structure contain valid data.

TITLE	DESCRIPTION
WHEA_PSHED_PLUGIN_CALLBACKS	The WHEA_PSHED_PLUGIN_CALLBACKS structure describes the callback functions for a PSHED plug-in.
WHEA_PSHED_PLUGIN_REGISTRATION_PACKET	The WHEA_PSHED_PLUGIN_REGISTRATION_PACKET structure describes the data required for registering a PSHED plug-in with the PSHED.
WHEA_REVISION	The WHEA_REVISION union describes the revision of the error record data structures.
WHEA_TIMESTAMP	The WHEA_TIMESTAMP union describes the time that an error was reported to the operating system.
WHEA_X64_REGISTER_STATE	The WHEA_X64_REGISTER_STATE structure describes the state of an x64 processor's registers.
WHEA_X86_REGISTER_STATE	The WHEA_X86_REGISTER_STATE structure describes the state of an x86 processor's registers.
WHEA_XPF_BUS_CHECK	The WHEA_XPF_BUS_CHECK union describes bus error information for an x86 or x64 processor.
WHEA_XPF_CACHE_CHECK	The WHEA_XPF_CACHE_CHECK union describes cache error information for an x86 or x64 processor.
WHEA_XPF_CMC_DESCRIPTOR	The WHEA_XPF_CMC_DESCRIPTOR structure describes a corrected machine check (CMC) error source for an x86 or x64 processor.
WHEA_XPF_CONTEXT_INFO	The WHEA_XPF_CONTEXT_INFO structure describes processor context information for an x86 or x64 processor.
WHEA_XPF_MC_BANK_DESCRIPTOR	The WHEA_XPF_MC_BANK_DESCRIPTOR structure describes a bank of machine check registers for an x86 or x64 processor.
WHEA_XPF_MCA_SECTION	
WHEA_XPF_MCE_DESCRIPTOR	The WHEA_XPF_MCE_DESCRIPTOR structure describes a machine check exception (MCE) error source for an x86 or x64 processor.
WHEA_XPF_MS_CHECK	The WHEA_XPF_MS_CHECK union describes microarchitecture-specific error information for an x86 or x64 processor.
WHEA_XPF_NMI_DESCRIPTOR	The WHEA_XPF_NMI_DESCRIPTOR structure describes a nonmaskable interrupt (NMI) error source for an x86 or x64 processor.
WHEA_XPF_PROCESSOR_ERROR_SECTION	The WHEA_XPF_PROCESSOR_ERROR_SECTION structure describes processor error data that is specific to the x86/x64 processor architecture.

TITLE	DESCRIPTION
WHEA_XPF_PROCESSOR_ERROR_SECTION_VALIDBITS	The WHEA_XPF_PROCESSOR_ERROR_SECTION_VALIDBITS union describes which members of a WHEA_XPF_PROCESSOR_ERROR_SECTION structure contain valid data and the number of structures that are contained in the WHEA_XPF_PROCESSOR_ERROR_SECTION structure's VariableInfo member.
WHEA_XPF_PROCINFO	The WHEA_XPF_PROCINFO structure describes processor error information that is specific to the x86 and x64 processor architectures.
WHEA_XPF_PROCINFO_VALIDBITS	The WHEA_XPF_PROCINFO_VALIDBITS union describes which members of a WHEA_XPF_PROCINFO structure contain valid data.
WHEA_XPF_TLB_CHECK	The WHEA_XPF_TLB_CHECK union describes translation lookaside buffer (TLB) error information for an x86 or x64 processor.

Enumerations

TITLE	DESCRIPTION
BDCB_CALLBACK_TYPE	The BDCB_CALLBACK_TYPE enumeration specifies whether the callback being passed to a BOOT_DRIVER_CALLBACK_FUNCTION routine is a status update or a boot-start driver initialization notification.
BDCB_CLASSIFICATION	The BDCB_CLASSIFICATION enumeration lists different classifications of boot start images.
BDCB_STATUS_UPDATE_TYPE	The BDCB_STATUS_UPDATE_TYPE enumeration lists the types of boot-driver callback status updates.
BUS_DATA_TYPE	The BUS_DATA_TYPE enumeration indicates the type of bus configuration space.
HAL_QUERY_INFORMATION_CLASS	The values are reserved for internal use.
HAL_SET_INFORMATION_CLASS	The values are reserved for internal use.
HARDWARE_COUNTER_TYPE	The HARDWARE_COUNTER_TYPE enumeration specifies the type of a hardware counter.
PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE	The PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE enumeration contains constants that indicate the type of hardware performance counter resource that is described by a PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure.
PROCESS_MITIGATION_POLICY	Represents the different process mitigation policies.
PSCREATEPROCESSNOTIFYTYPE	Indicates the type of process notification. This enumeration is used in PsSetCreateProcessNotifyRoutineEx2 to register callback notifications.

TITLE	DESCRIPTION
PSCREATETHREADNOTIFYTYPE	Indicates the type of thread notification. This enumeration is used in PsSetCreateThreadNotifyRoutineEx to register callback notifications associated with thread creation or deletion.
SOC_SUBSYSTEM_TYPE	The SOC_SYBSYSTEM_TYPE enumeration defines different types of System on a Chip (SoC) subsystems.
STATE_LOCATION_TYPE	Defines values for persisted state locations for device and driver files.
SUBSYSTEM_INFORMATION_TYPE	Indicates the type of subsystem for a process or thread. This enumeration is used in NtQueryInformationProcess and NtQueryInformationThread calls.
WELL_KNOWN_SID_TYPE	A list of commonly used security identifiers (SIDs). Programs can pass these values to the **CreateWellKnownSid** function to create a SID from this list. For information about these values, see WELL_KNOWN_SID_TYPE.
WHEA_ERROR_PACKET_DATA_FORMAT	The WHEA_ERROR_PACKET_DATA_FORMAT enumeration defines the raw hardware error data format in a hardware error packet.
WHEA_ERROR_SEVERITY	The WHEA_ERROR_SEVERITY enumeration defines the possible severity levels of a hardware error condition.
WHEA_ERROR_SOURCE_STATE	The WHEA_ERROR_SOURCE_STATE enumeration defines the different runtime states for an error source.
WHEA_ERROR_SOURCE_TYPE	The WHEA_ERROR_SOURCE_TYPE enumeration defines the different types of error sources that can report hardware errors.
WHEA_ERROR_TYPE	The WHEA_ERROR_TYPE enumeration defines the different types of hardware components that can report a hardware error.
WHEA_RAW_DATA_FORMAT	The WHEA_RAW_DATA_FORMAT enumeration defines the possible formats that raw hardware error data can be encoded in a hardware error packet.

BDCB_CALLBACK_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The BDCB_CALLBACK_TYPE enumeration specifies whether the callback being passed to a BOOT_DRIVER_CALLBACK_FUNCTION routine is a status update or a boot-start driver initialization notification.

Syntax

```
typedef enum _BDCB_CALLBACK_TYPE {
  BdCbStatusUpdate,
  BdCbInitializeImage
} BDCB_CALLBACK_TYPE, *PBDCB_CALLBACK_TYPE;
```

Constants

BdCbStatusUpdate	A status update provided by the system to a boot-start driver.
BdCbInitializeImage	A boot image is about to be initialized. During this callback, boot-start drivers may classify a boot image as a known good image or a known bad image.

Remarks

The two callback types have unique context structures that provide additional information specific to the callback.

VALUE	CORRESPONDING STRUCTURE TO USE
BdCbStatusUpdate	BDCB_STATUS_UPDATE_TYPE
BdCbInitializeImage	BDCB_CLASSIFICATION

Requirements

Minimum supported client	Available starting with Windows 8.
Header	ntddk.h (include Ntddk.h)

See Also

BDCB_CLASSIFICATION

BDCB_STATUS_UPDATE_TYPE

BOOT_DRIVER_CALLBACK_FUNCTION

BDCB_CLASSIFICATION Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The BDCB_CLASSIFICATION enumeration lists different classifications of boot start images.

Syntax

```
typedef enum _BDCB_CLASSIFICATION {
  BdCbClassificationUnknownImage,
  BdCbClassificationKnownGoodImage,
  BdCbClassificationKnownBadImage,
  BdCbClassificationKnownBadImageBootCritical,
  BdCbClassificationEnd
} BDCB_CLASSIFICATION, *PBDCB_CLASSIFICATION;
```

Constants

BdCbClassificationUnknownImage	The boot start image has not been inspected by anti-malware or anti-malware does not have enough information to determine whether the binary is malware.
BdCbClassificationKnownGoodImage	The boot start image has been inspected by anti-malware and found not to be malware.
BdCbClassificationKnownBadImage	The boot start image has been inspected by anti-malware and found to be malware.
BdCbClassificationKnownBadImageBootCritical	The boot start image has been inspected by anti-malware and found to be malware, but the anti-malware boot-start driver also knows it to be critical to the success of the boot.
BdCbClassificationEnd	Do not use. Reserved for future use.

Requirements

Minimum supported client	Available starting with Windows 8.
Header	ntddk.h (include Ntddk.h)

See Also

BDCB_IMAGE_INFORMATION

BOOT_DRIVER_CALLBACK_FUNCTION

BDCB_IMAGE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **BDCB_IMAGE_INFORMATION** structure describes information about a boot-start driver that is about to be initialized, provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.

Syntax

```
typedef struct _BDCB_IMAGE_INFORMATION {
  BDCB_CLASSIFICATION Classification;
  ULONG               ImageFlags;
  UNICODE_STRING      ImageName;
  UNICODE_STRING      RegistryPath;
  UNICODE_STRING      CertificatePublisher;
  UNICODE_STRING      CertificateIssuer;
  PVOID               ImageHash;
  PVOID               CertificateThumbprint;
  ULONG               ImageHashAlgorithm;
  ULONG               ThumbprintHashAlgorithm;
  ULONG               ImageHashLength;
  ULONG               CertificateThumbprintLength;
} BDCB_IMAGE_INFORMATION, *PBDCB_IMAGE_INFORMATION;
```

Members

`Classification`

The classification of the boot start image.

`ImageFlags`

Bit flags that describe the image. The following values are defined.

VALUE	MEANING
Bit 0	The boot start image is a dependent DLL.
Bit 1	The boot start image failed code integrity but load was allowed because of boot policy (code integrity not required on the platform, or code integrity disabled because of boot setting, debugging, or both).
Bits 2-31	Do not use. Reserved.

`ImageName`

The name of the boot-start driver's binary image.

`RegistryPath`

The path in the registry where the boot-start driver is registered.

`CertificatePublisher`

The publisher of the image's certificate. If the image is not signed, the string is empty ("").

`CertificateIssuer`

The issuer of the image's certificate. If the image is not signed, the string is empty ("").

`ImageHash`

The Authenticode hash of the image, which can be calculated by using SignTool.exe (Sign Tool).

`CertificateThumbprint`

The hash of the certificate of the signer to be signed. Run **certutil –dump x,cer** to view this value as "Signature Hash".

`ImageHashAlgorithm`

The algorithm of the image hash. The following values are listed for reference.

VALUE
ALG_CLASS_HASH
ALG_CLASS_ANY
ALG_SID_MD5
ALG_SID_SHA1
ALG_SID_SHA_256
ALG_SID_SHA_384
ALG_SID_SHA_512
CALG_MD5
CALG_SHA1
CALG_SHA_256
CALG_SHA_384
CALG_SHA_512

`ThumbprintHashAlgorithm`

The algorithm of the certificate thumbprint. This member should be ignored if **CertificateThumbprint** is NULL.

`ImageHashLength`

The length of data pointed to by the **ImageHash** member.

`CertificateThumbprintLength`

The length of data pointed to by the **CertificateThumbprint** member.

Requirements

Minimum supported client	Available starting with Windows 8.
Header	ntddk.h (include Ntddk.h)

See Also

BDCB_CLASSIFICATION

BOOT_DRIVER_CALLBACK_FUNCTION

BDCB_STATUS_UPDATE_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The BDCB_STATUS_UPDATE_CONTEXT structure describes a status update provided by Windows to a boot-start driver's BOOT_DRIVER_CALLBACK_FUNCTION routine.

Syntax

```
typedef struct _BDCB_STATUS_UPDATE_CONTEXT {
  BDCB_STATUS_UPDATE_TYPE StatusType;
} BDCB_STATUS_UPDATE_CONTEXT, *PBDCB_STATUS_UPDATE_CONTEXT;
```

Members

`StatusType`

The type of the status update.

Remarks

Boot-start drivers must not unregister their boot-start driver callbacks during a callback. Doing so may result in a bug check. To properly unregister a boot-driver callback, boot-start drivers must specify an Unload routine in their driver object and call IoUnRegisterBootDriverCallback from within the Unload dispatch routine.

Requirements

Minimum supported client	Available starting with Windows 8.
Header	ntddk.h (include Ntddk.h)

See Also

BDCB_STATUS_UPDATE_TYPE

BOOT_DRIVER_CALLBACK_FUNCTION

IoUnRegisterBootDriverCallback

BDCB_STATUS_UPDATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The BDCB_STATUS_UPDATE_TYPE enumeration lists the types of boot-driver callback status updates.

Syntax

```
typedef enum _BDCB_STATUS_UPDATE_TYPE {
  BdCbStatusPrepareForDependencyLoad,
  BdCbStatusPrepareForDriverLoad,
  BdCbStatusPrepareForUnload
} BDCB_STATUS_UPDATE_TYPE, *PBDCB_STATUS_UPDATE_TYPE;
```

Constants

BdCbStatusPrepareForDependencyLoad	Windows will start loading driver dependencies next.
BdCbStatusPrepareForDriverLoad	Windows has completed loading driver dependencies and will start loading boot-start drivers.
BdCbStatusPrepareForUnload	Windows has completed the initialization of all boot-start drivers. After the completion of this callback, the Boot Driver Callback Facility will be torn down and no more callbacks will be received. During this callback, Early Launch AM drivers must clean up and prepare to be unloaded.

Requirements

Minimum supported client	Available starting with Windows 8.
Header	ntddk.h (include Ntddk.h)

See Also

BDCB_STATUS_UPDATE_CONTEXT

BOOT_DRIVER_CALLBACK_FUNCTION

BUS_DATA_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **BUS_DATA_TYPE** enumeration indicates the type of bus configuration space.

Syntax

```
typedef enum _BUS_DATA_TYPE {
  ConfigurationSpaceUndefined,
  Cmos,
  EisaConfiguration,
  Pos,
  CbusConfiguration,
  PCIConfiguration,
  VMEConfiguration,
  NuBusConfiguration,
  PCMCIAConfiguration,
  MPIConfiguration,
  MPSAConfiguration,
  PNPISAConfiguration,
  SgiInternalConfiguration,
  MaximumBusDataType
} BUS_DATA_TYPE, *PBUS_DATA_TYPE;
```

Constants

ConfigurationSpaceUndefined	Indicates that the type of bus configuration space is undefined.
Cmos	Indicates CMOS data.
EisaConfiguration	Indicates an EISA bus configuration space.
Pos	For internal use only.
CbusConfiguration	Indicates Cbus configuration space.
PCIConfiguration	Indicates PCI configuration space.
VMEConfiguration	Indicates VME configuration space.
NuBusConfiguration	Indicates NuBus configuration space.
PCMCIAConfiguration	Indicates PCMCIA configuration space.
MPIConfiguration	Indicates MPI configuration space.
MPSAConfiguration	Indicates MPSA configuration space.
PNPISAConfiguration	Indicates PNPISA configuration space.

SgiInternalConfiguration	Indicates SGI internal bus configuration space.
MaximumBusDataType	Indicates the upper limit of the bus configuration space types.

Requirements

Header	ntddk.h (include Ntddk.h, Miniport.h)

See Also

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

CONTROLLER_OBJECT structure

1/30/2019 • 2 minutes to read • Edit Online

A controller object represents a hardware adapter or controller with homogenous devices that are the actual targets for I/O requests. A controller object can be used to synchronize a device driver's I/O to the target devices through its hardware adapter/controller.

A controller object is partially opaque. Driver writers must know about a certain field associated with the controller object because their drivers access this field through the controller object pointer returned by **IoCreateController**. The following field in a controller object is accessible to the creating driver.

Syntax

```
typedef struct _CONTROLLER_OBJECT {
  CSHORT         Type;
  CSHORT         Size;
  PVOID          ControllerExtension;
  KDEVICE_QUEUE DeviceWaitQueue;
  ULONG          Spare1;
  LARGE_INTEGER Spare2;
} CONTROLLER_OBJECT, *PCONTROLLER_OBJECT;
```

Members

`Type`

`Size`

`ControllerExtension`

Pointer to the controller extension. The structure and contents of the controller extension are driver-defined. The size is driver-determined, specified in the driver's call to **IoCreateController**. Usually, drivers maintain common state about I/O operations in the controller extension and device-specific state about I/O for a target device in the corresponding device extension.

`DeviceWaitQueue`

`Spare1`

`Spare2`

Remarks

Most driver routines that process IRPs are given a pointer to the target device object. Consequently, device drivers that use controller objects frequently store the controller object pointer returned by **IoCreateController** in each device extension.

Note that a controller object has no name so it cannot be the target of an I/O request, and higher-level drivers cannot connect or attach their device objects to a device driver's controller object.

Undocumented fields within a controller object should be considered inaccessible. Drivers with dependencies on object field locations or access to undocumented fields might not remain portable and interoperable with other drivers over time.

Requirements

Header	ntddk.h (include Ntddk.h)

See Also

IoCreateController

FILE_ALIGNMENT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_ALIGNMENT_INFORMATION** structure is used as an argument to the ZwQueryInformationFile routine.

Syntax

```
typedef struct _FILE_ALIGNMENT_INFORMATION {
  ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFORMATION, *PFILE_ALIGNMENT_INFORMATION;
```

Members

`AlignmentRequirement`

The buffer alignment required by the underlying device. For a list of system-defined values, see DEVICE_OBJECT. The value must be one of the FILE_*XXX*_ALIGNMENT values defined in Wdm.h. For more information, see DEVICE_OBJECT and Initializing a Device Object.

Requirements

Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

DEVICE_OBJECT

ZwQueryInformationFile

ZwSetInformationFile

FILE_ATTRIBUTE_TAG_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_ATTRIBUTE_TAG_INFORMATION** structure is used as an argument to ZwQueryInformationFile.

Syntax

```
typedef struct _FILE_ATTRIBUTE_TAG_INFORMATION {
  ULONG FileAttributes;
  ULONG ReparseTag;
} FILE_ATTRIBUTE_TAG_INFORMATION, *PFILE_ATTRIBUTE_TAG_INFORMATION;
```

Members

`FileAttributes`

Specifies one or more FILE_ATTRIBUTE_*XXX* flags. For descriptions of these flags, see the documentation of the **GetFileAttributes** function in the Microsoft Windows SDK.

`ReparseTag`

Specifies the reparse point tag. If the **FileAttributes** member includes the FILE_ATTRIBUTE_REPARSE_POINT attribute flag, this member specifies the reparse tag. Otherwise, this member is unused.

Remarks

FILE_READ_ATTRIBUTES access to the file is required to query this information.

For more information about reparse points and reparse point tags, see the Windows SDK.

This structure must be aligned on a LONG (4-byte) boundary.

Requirements

Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

ZwQueryInformationFile

FILE_DISPOSITION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_DISPOSITION_INFORMATION** structure is used as an argument to the ZwSetInformationFile routine.

Syntax

```
typedef struct _FILE_DISPOSITION_INFORMATION {
  BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION, *PFILE_DISPOSITION_INFORMATION;
```

Members

`DeleteFile`

Indicates whether the operating system file should delete the file when the file is closed. Set this member to **TRUE** to delete the file when it is closed. Otherwise, set to **FALSE**. Setting this member to **FALSE** has no effect if the handle was opened with FILE_FLAG_DELETE_ON_CLOSE.

Remarks

The caller must have DELETE access to a given file in order to call **ZwSetInformationFile** with **DeleteFile** set to **TRUE** in this structure. Subsequently, the only legal operation by such a caller is to close the open file handle.

A file marked for deletion is not actually deleted until all open handles for the file object have been closed and the link count for the file is zero.

Requirements

Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

ZwClose

ZwSetInformationFile

FILE_END_OF_FILE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_END_OF_FILE_INFORMATION** structure is used as an argument to the ZwSetInformationFile routine.

Syntax

```
typedef struct _FILE_END_OF_FILE_INFORMATION {
  LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFORMATION, *PFILE_END_OF_FILE_INFORMATION;
```

Members

`EndOfFile`

The absolute new end of file position as a byte offset from the start of the file.

Remarks

EndOfFile specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file: that is, it is the offset to the byte immediately following the last valid byte in the file.

Requirements

Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

ZwQueryInformationFile

ZwSetInformationFile

FILE_NAME_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_NAME_INFORMATION** structure is used as argument to the ZwQueryInformationFile and ZwSetInformationFile routines.

Syntax

```
typedef struct _FILE_NAME_INFORMATION {
  ULONG FileNameLength;
  WCHAR FileName[1];
} FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
```

Members

`FileNameLength`

Specifies the length, in bytes, of the file name string.

`FileName`

Specifies the first character of the file name string. This is followed in memory by the remainder of the string.

Remarks

The **ZwQueryInformationFile** routine uses this structure to return the file name string to the caller. For more information about the form of the name returned, see ZwQueryInformationFile.

Callers of ZwSetInformationFile can use this structure to specify a new short name for a file.

Requirements

Header	ntddk.h (include Ntddk.h)

See Also

ZwQueryInformationFile

ZwSetInformationFile

FILE_VALID_DATA_LENGTH_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_VALID_DATA_LENGTH_INFORMATION** structure is used as an argument to ZwSetInformationFile.

Syntax

```
typedef struct _FILE_VALID_DATA_LENGTH_INFORMATION {
  LARGE_INTEGER ValidDataLength;
} FILE_VALID_DATA_LENGTH_INFORMATION, *PFILE_VALID_DATA_LENGTH_INFORMATION;
```

Members

`ValidDataLength`

Specifies the new valid data length for the file. This parameter must be a positive value that is greater than the current valid data length, but less than or equal to the current file size.

Remarks

The **FILE_VALID_DATA_LENGTH_INFORMATION** structure is used to set a new valid data length for a file on an NTFS volume. A file's *valid data length* is the length, in bytes, of the data that has been written to the file. This *valid data* extends from the beginning of the file to the last byte in the file that has not been zeroed or left uninitialized.

Setting this information requires FILE_WRITE_DATA access to the file. In addition, nonadministrators and remote users must have **SeManageVolumePrivilege** (SE_MANAGE_VOLUME_PRIVILEGE) for the volume on which the file resides.

File system filter drivers can find it useful to set a valid data length in the following scenarios:

- When writing raw clusters directly to disk through a hardware channel. This allows the filter driver to inform the file system that this range contains valid data that can be returned to the user.
- When creating large files where performance is an issue. This avoids the time it takes to fill the file with zeros when the file is created or extended.
- When remotely extending a file and writing to disk on a served-metadata cluster file system.

The size of the *FileInformation* buffer passed to **ZwSetInformationFile** must be at least **sizeof**(**FILE_VALID_DATA_LENGTH_INFORMATION**).

This structure must be aligned on a LONGLONG (8-byte) boundary.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.

Header	ntddk.h (include Ntddk.h)

See Also

ZwSetInformationFile

HARDWARE_COUNTER structure

1/30/2019 • 2 minutes to read • Edit Online

The **HARDWARE_COUNTER** structure contains information about a hardware counter.

Syntax

```
typedef struct _HARDWARE_COUNTER {
  HARDWARE_COUNTER_TYPE Type;
  ULONG                 Reserved;
  ULONG64               Index;
} HARDWARE_COUNTER, *PHARDWARE_COUNTER;
```

Members

`Type`

Specifies the type of the hardware counter. Set this member to the following HARDWARE_COUNTER_TYPE enumeration value:

- **PMCCounter**

`Reserved`

Reserved for use by the operating system. Initialize this member to zero.

`Index`

Specifies the hardware counter index. Each hardware counter in a performance monitoring unit (PMU) for a processor is identified by an index.

Remarks

This structure is used by the KeQueryHardwareCounterConfiguration and KeSetHardwareCounterConfiguration routines.

The **Type** member specifies the type of hardware counter that is described by the structure. In Windows 7, the only defined hardware counter type is **PMCCounter**, which is a performance monitor counter. This type of counter is used by thread-profiling applications.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	ntddk.h (include Ntddk.h)

See Also

HARDWARE_COUNTER_TYPE

KeQueryHardwareCounterConfiguration

KeSetHardwareCounterConfiguration

HARDWARE_COUNTER_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **HARDWARE_COUNTER_TYPE** enumeration specifies the type of a hardware counter.

Syntax

```
typedef enum _HARDWARE_COUNTER_TYPE {
  PMCCounter,
  MaxHardwareCounterType
} HARDWARE_COUNTER_TYPE, *PHARDWARE_COUNTER_TYPE;
```

Constants

PMCCounter	Performance monitor counter. This type of counter is used by thread-profiling applications.
MaxHardwareCounterType	The maximum value in this enumeration type.

Remarks

The **Type** member of the HARDWARE_COUNTER structure contains a **HARDWARE_COUNTER_TYPE** enumeration value.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	ntddk.h (include Winnt.h, Ntddk.h)

See Also

HARDWARE_COUNTER

IMAGE_INFO structure

2/5/2019 • 2 minutes to read • Edit Online

Used by driver's load-image routine (PLOAD_IMAGE_NOTIFY_ROUTINE) to specify image information.

Syntax

```
typedef struct _IMAGE_INFO {
  union {
    ULONG Properties;
    struct {
      ULONG ImageAddressingMode : 8;
      ULONG SystemModeImage : 1;
      ULONG ImageMappedToAllPids : 1;
      ULONG ExtendedInfoPresent : 1;
      ULONG MachineTypeMismatch : 1;
      ULONG ImageSignatureLevel : 4;
      ULONG ImageSignatureType : 3;
      ULONG ImagePartialMap : 1;
      ULONG Reserved : 12;
    };
  };
  PVOID  ImageBase;
  ULONG  ImageSelector;
  SIZE_T ImageSize;
  ULONG  ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Members

`Properties`

`ImageAddressingMode`

Always set to IMAGE_ADDRESSING_MODE_32BIT.

`SystemModeImage`

Set either to one for newly loaded kernel-mode components, such as drivers, or to zero for images that are mapped into user space.

`ImageMappedToAllPids`

Always set to zero.

`ExtendedInfoPresent`

If the **ExtendedInfoPresent** flag is set, the **IMAGE_INFO** structure is part of a larger, extended version of the image information structure (see IMAGE_INFO_EX). Added in Windows Vista. For more information, see "Extended version of the image information structure" later in this Remarks section.

`MachineTypeMismatch`

Always set to zero. Added in Windows 8/Windows Server 2012.

`ImageSignatureLevel`

The level of signature with which code integrity has labeled the image. This value is one of the `#define SE_SIGNING_LEVEL_*` constants in ntddk.h. Added in Windows 8.1/Windows Server 2012 R2.

`ImageSignatureType`

The type of signature with which code integrity has labeled the image. This value is a **SE_IMAGE_SIGNATURE_TYPE** enumeration value, defined in ntddk.h. Added in Windows 8.1/Windows Server 2012 R2.

`ImagePartialMap`

This value is non-zero if the image mapping view called out for is a partial view that does not map the entire image; 0 if the view maps the entire image. Added in Windows 10/Windows Server 2016.

`Reserved`

Always set to zero.

`ImageBase`

Set to the virtual base address of the image.

`ImageSelector`

Always set to zero.

`ImageSize`

Set to the virtual size, in bytes, of the image.

`ImageSectionNumber`

Always set to zero.

Remarks

If the **ExtendedInfoPresent** flag is set, the **IMAGE_INFO** structure is part of a larger, extended version of the image information structure, IMAGE_INFO_EX.

Requirements

Header	ntddk.h (include Ntddk.h)

See Also

IMAGE_INFO_EX

PLOAD_IMAGE_NOTIFY_ROUTINE

PsSetLoadImageNotifyRoutine

IMAGE_INFO_EX structure

1/30/2019 • 2 minutes to read • Edit Online

Extended version of the image information structure (see IMAGE_INFO).

Syntax

```
typedef struct _IMAGE_INFO_EX {
  SIZE_T               Size;
  IMAGE_INFO           ImageInfo;
  struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

Members

`Size`

Specifies the size, in bytes, of the **IMAGE_INFO_EX** structure.

`ImageInfo`

An IMAGE_INFO structure that specifies image information.

`FileObject`

Contains a pointer to the file object of the backing file for the image. The driver can take a reference to this object or use it for other operations.

Remarks

If the **ExtendedInfoPresent** flag is set, the **IMAGE_INFO** structure is part of a larger, extended version of the image information structure. In this case, the load-image notify routine can use the **CONTAINING_RECORD** macro in the Winnt.h header file to obtain the base address of the **IMAGE_INFO_EX** structure.

Requirements

Header	ntddk.h (include Ntddk.h)

See Also

IMAGE_INFO

PLOAD_IMAGE_NOTIFY_ROUTINE

PsSetLoadImageNotifyRoutine

KEY_CACHED_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_CACHED_INFORMATION** structure holds the cached information available for a registry key or subkey.

Syntax

```
typedef struct _KEY_CACHED_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG         TitleIndex;
  ULONG         SubKeys;
  ULONG         MaxNameLen;
  ULONG         Values;
  ULONG         MaxValueNameLen;
  ULONG         MaxValueDataLen;
  ULONG         NameLength;
} KEY_CACHED_INFORMATION, *PKEY_CACHED_INFORMATION;
```

Members

`LastWriteTime`

The last time the key or any of its values changed. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

`TitleIndex`

Device and intermediate drivers should ignore this member.

`SubKeys`

The number of subkeys for a key.

`MaxNameLen`

The maximum number of bytes for a subkey name.

`Values`

The number of value entries.

`MaxValueNameLen`

The maximum length, in bytes, of any value entry name.

`MaxValueDataLen`

The maximum length, in bytes, of a value entry data field.

`NameLength`

The size, in bytes, of the key name.

Requirements

Minimum supported client	Available in Windows Vista and later versions of the Windows operating system.
Header	ntddk.h (include Ntddk.h)

See Also

KEY_BASIC_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwEnumerateKey

ZwQueryKey

KEY_NAME_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_NAME_INFORMATION** structure holds the name and name length of the key.

Syntax

```
typedef struct _KEY_NAME_INFORMATION {
  ULONG NameLength;
  WCHAR Name[1];
} KEY_NAME_INFORMATION, *PKEY_NAME_INFORMATION;
```

Members

`NameLength`

The size, in bytes, of the key name string in the **Name** array.

`Name`

An array of wide characters that contains the name of the key. This character string is not null-terminated. Only the first element in this array is included in the **KEY_NAME_INFORMATION** structure definition. The storage for the remaining elements in the array immediately follows this element.

Remarks

The ZwQueryKey routine uses the **KEY_NAME_INFORMATION** structure to contain the registry key name. When the *KeyInformationClass* parameter of this routine is **KeyNameInformation**, the *KeyInformation* buffer is treated as a **KEY_NAME_INFORMATION** structure. For more information about the **KeyNameInformation** enumeration value, see KEY_INFORMATION_CLASS.

Requirements

Header	ntddk.h (include Ntddk.h)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_INFORMATION_CLASS

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwQueryKey

KEY_VIRTUALIZATION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VIRTUALIZATION_INFORMATION** structure defines the basic information that is available for a registry key or subkey.

Syntax

```
typedef struct _KEY_VIRTUALIZATION_INFORMATION {
  ULONG VirtualizationCandidate : 1;
  ULONG VirtualizationEnabled : 1;
  ULONG VirtualTarget : 1;
  ULONG VirtualStore : 1;
  ULONG VirtualSource : 1;
  ULONG Reserved : 27;
} KEY_VIRTUALIZATION_INFORMATION, *PKEY_VIRTUALIZATION_INFORMATION;
```

Members

`VirtualizationCandidate`

Specifies whether the key is part of the virtualization namespace scope.

`VirtualizationEnabled`

Specifies whether virtualization is enabled on this key. This value can be set to 1 only if **VirtualizationCandidate** is 1.

`VirtualTarget`

Specifies whether the key is a virtual key. This value can be set to 1 only if **VirtualizationCandidate** and **VirtualizationEnabled** are both 0. This value is valid only on the virtual store key handles.

`VirtualStore`

Specified whether the key is a part of the virtual store path.

`VirtualSource`

Specifies whether the key has ever been virtualized. This value can be set to 1 only if **VirtualizationCandidate** is 1.

`Reserved`

This value is reserved for system use.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system.

Header	ntddk.h (include Ntddk.h)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

ZwEnumerateKey

ZwQueryKey

MM_COPY_ADDRESS structure

2/5/2019 • 2 minutes to read • Edit Online

The **MM_COPY_ADDRESS** structure contains either a virtual memory address or a physical memory address.

Syntax

```
typedef struct _MM_COPY_ADDRESS {
  union {
    PVOID            VirtualAddress;
    PHYSICAL_ADDRESS PhysicalAddress;
  };
} MM_COPY_ADDRESS, *PMMCOPY_ADDRESS;
```

Members

`VirtualAddress`

`PhysicalAddress`

Remarks

This structure is used by the MmCopyMemory routine.

Requirements

Minimum supported client	Supported starting with Windows 8.1.
Header	ntddk.h

See Also

MmCopyMemory

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure

1/30/2019 • 2 minutes to read • Edit Online

The **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure describes the counter resources available on the platform.

Syntax

```
typedef struct _PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR {
  PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE Type;
  ULONG                                     Flags;
  union {
    ULONG CounterIndex;
    ULONG ExtendedRegisterAddress;
    struct {
      ULONG Begin;
      ULONG End;
    } Range;
  } u;
} PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR, *PPHYSICAL_COUNTER_RESOURCE_DESCRIPTOR;
```

Members

`Type`

Indicates the type of counter resource described by the structure. Set this parameter to one of the following PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE enumeration constants:

- **ResourceTypeSingle**
- **ResourceTypeRange**
- **ResourceTypeExtendedCounterConfiguration**
- **ResourceTypeOverflow**

`Flags`

No flags are currently defined. Set this member to zero.

`u`

A union that can contain a **CounterIndex**, **ExtendedRegisterAddress**, or **Range** value.

`u.CounterIndex`

A physical counter index.

`u.ExtendedRegisterAddress`

An extended configuration register address.

`u.Range`

`u.Range.Begin`

The starting counter index of a block of contiguous counters.

`u.Range.End`

The ending counter index of a block of contiguous counters.

Remarks

Most processors have performance monitor units that contain a number of hardware counters for measuring various aspects of system performance. The **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure describes a counter resource, which can be a single hardware counter, a block of contiguous counters, or a counter overflow interrupt. This structure's **Type** member indicates which type of counter resource is described by the structure.

To specify a set of counter resources, a kernel-mode client can provide an array of **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structures. A PHYSICAL_COUNTER_RESOURCE_LIST structure serves as a header for the array. The client can allocate and free the set of counter resources described by the array as a single unit.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE

PHYSICAL_COUNTER_RESOURCE_LIST

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE** enumeration contains constants that indicate the type of hardware performance counter resource that is described by a PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structure.

Syntax

```
typedef enum _PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE {
  ResourceTypeSingle,
  ResourceTypeRange,
  ResourceTypeExtendedCounterConfiguration,
  ResourceTypeOverflow,
  ResourceTypeMax
} PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE;
```

Constants

ResourceTypeSingle	A single hardware counter. The counter is described by the **u.CounterIndex** member of the **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure.
ResourceTypeRange	A range of counter indexes. The counter indexes are described by the **u.Range** member of the **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure.
ResourceTypeExtendedCounterConfiguration	An extended counter configuration register address. The register address is contained in the **u.ExtendedRegisterAddress** member of the **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure. This value is used only on Intel NetBurst systems.
ResourceTypeOverflow	A counter overflow interrupt. The **u** member of the **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure is not used for this counter resource type.
ResourceTypeMax	The maximum value in this enumeration type.

Remarks

The **Type** member of a **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structure uses a **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR_TYPE** enumeration constant to indicate the type of counter resource that is described by the structure.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR

PHYSICAL_COUNTER_RESOURCE_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **PHYSICAL_COUNTER_RESOURCE_LIST** structure describes an array of PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR structures.

Syntax

```
typedef struct _PHYSICAL_COUNTER_RESOURCE_LIST {
  ULONG                                Count;
  PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR Descriptors[ANYSIZE_ARRAY];
} PHYSICAL_COUNTER_RESOURCE_LIST, *PPHYSICAL_COUNTER_RESOURCE_LIST;
```

Members

`Count`

The number of elements in the **Descriptors** array.

`Descriptors`

The first element in an array of **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** structures. If the array contains more than one element, the remaining elements immediately follow the first element.

Remarks

This structure describes a set of counter resources. Most processors have performance monitor units (PMUs) that contain a number of hardware counters to measure various aspects of system performance. A counter resource is a single hardware counter, a block of contiguous counters, or a counter overflow interrupt in a PMU.

The buffer allocated to contain a **PHYSICAL_COUNTER_RESOURCE_LIST** structure must be large enough to contain the structure plus any **PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR** array elements that follow the structure.

The HalAllocateHardwareCounters routine's *ResourceList* parameter is a pointer to a **PHYSICAL_COUNTER_RESOURCE_LIST** structure. In Windows 7, this parameter is unused and must be set to **NULL**.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of Windows.
Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

HalAllocateHardwareCounters

PHYSICAL_COUNTER_RESOURCE_DESCRIPTOR

PNP_LOCATION_INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PNP_LOCATION_INTERFACE** structure describes the GUID_PNP_LOCATION_INTERFACE interface.

Syntax

```
typedef struct _PNP_LOCATION_INTERFACE {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
  PGET_LOCATION_STRING   GetLocationString;
} PNP_LOCATION_INTERFACE, *PPNP_LOCATION_INTERFACE;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined version of the interface.

`Context`

A pointer to interface-specific context information.

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the reference count for the interface. The PINTERFACE_REFERENCE function pointer type is defined in the Wdm.h header file.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the reference count for the interface. The PINTERFACE_DEREFERENCE function pointer type is defined in the Wdm.h header file.

`GetLocationString`

A pointer to the PnpGetLocationString routine for the interface. The routine supplies the device-specific part of the SPDRP_LOCATION_PATHS property for the device.

Remarks

A driver obtains a pointer to the **PNP_LOCATION_INTERFACE** structure by sending an IRP_MN_QUERY_INTERFACE IRP to its bus driver with **InterfaceType** set to GUID_PNP_LOCATION_INTERFACE.

Requirements

Minimum supported client	Windows Server 2003 and later versions of the Windows operating system.
Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

PnpGetLocationString

POWER_THROTTLING_PROCESS_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the throttling policies and how to apply them to a target process when that process is subject to power management.

Syntax

```
typedef struct _POWER_THROTTLING_PROCESS_STATE {
  ULONG Version;
  ULONG ControlMask;
  ULONG StateMask;
} POWER_THROTTLING_PROCESS_STATE, *PPOWER_THROTTLING_PROCESS_STATE;
```

Members

`Version`

The version of this structure. Set to PROCESS_POWER_THROTTLING_CURRENT_VERSION.

`ControlMask`

Flags that enable the caller to take control of the power throttling mechanism.

- PROCESS_POWER_THROTTLING_EXECUTION_SPEED: Manages the execution speed of the process.

`StateMask`

Flags that manage the power throttling mechanism on/off state.

- PROCESS_POWER_THROTTLING_EXECUTION_SPEED: Manages the execution speed of the process.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ntddk.h

POWER_THROTTLING_THREAD_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the throttling policies and how to apply them to a target thread when that thread is subject to power management.

Syntax

```
typedef struct _POWER_THROTTLING_THREAD_STATE {
  ULONG Version;
  ULONG ControlMask;
  ULONG StateMask;
} POWER_THROTTLING_THREAD_STATE, *PPOWER_THROTTLING_THREAD_STATE;
```

Members

`Version`

The version of this structure. Set to THREAD_POWER_THROTTLING_CURRENT_VERSION.

`ControlMask`

Flags that enable the caller to take control of the power throttling mechanism.

- THREAD_POWER_THROTTLING_EXECUTION_SPEED: Manages the execution speed of the process.

`StateMask`

Flags that manage the power throttling mechanism on/off state.

- THREAD_POWER_THROTTLING_EXECUTION_SPEED: Manages the execution speed of the process.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ntddk.h

PROCESS_MITIGATION_CHILD_PROCESS_POLICY structure

1/30/2019 • 2 minutes to read • Edit Online

Stores policy information about creating child processes.

Syntax

```
typedef struct _PROCESS_MITIGATION_CHILD_PROCESS_POLICY {
  union {
    ULONG Flags;
    struct {
      ULONG NoChildProcessCreation : 1;
      ULONG AuditNoChildProcessCreation : 1;
      ULONG AllowSecureProcessCreation : 1;
      ULONG ReservedFlags : 29;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_CHILD_PROCESS_POLICY, *PPROCESS_MITIGATION_CHILD_PROCESS_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.NoChildProcessCreation`

If set, the process cannot create child processes.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditNoChildProcessCreation`

If set, causes audit events to be generated when child processes are created by the process. If both **NoChildProcessCreation** and **AuditNoChildProcessCreation** are set, **NoChildProcessCreation** takes precedence over audit setting.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AllowSecureProcessCreation`

Denies creation of child processes unless the child process is a secure process and if creation was previously blocked. It allows a process to spawn a child process on behalf of another process that cannot itself create child processes. See PROCESS_CREATION_CHILD_PROCESS_OVERRIDE in UpdateProcThreadAttribute.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved.

Requirements

Minimum supported client	Windows 10, version 1709

Minimum supported server	Windows Server 2016
Header	ntddk.h

PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY structure

1/30/2019 • 2 minutes to read • Edit Online

Stores information about process mitigation policy.

Syntax

```
typedef struct _PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY {
  union {
    ULONG Flags;
    struct {
      ULONG EnableExportAddressFilter : 1;
      ULONG AuditExportAddressFilter : 1;
      ULONG EnableExportAddressFilterPlus : 1;
      ULONG AuditExportAddressFilterPlus : 1;
      ULONG EnableImportAddressFilter : 1;
      ULONG AuditImportAddressFilter : 1;
      ULONG EnableRopStackPivot : 1;
      ULONG AuditRopStackPivot : 1;
      ULONG EnableRopCallerCheck : 1;
      ULONG AuditRopCallerCheck : 1;
      ULONG EnableRopSimExec : 1;
      ULONG AuditRopSimExec : 1;
      ULONG ReservedFlags : 20;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY, *PPROCESS_MITIGATION_PAYLOAD_RESTRICTION_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

Bitwise of flags in this structure.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableExportAddressFilter`

If set this enables the Export Address Filter mitigation in enforcement mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditExportAddressFilter`

If set this enables the Export Address Filter mitigation in audit mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableExportAddressFilterPlus`

If set this enables the Export Address Filter Plus mitigation in enforcement mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditExportAddressFilterPlus`

If set this enables the Export Address Filter mitigation in audit mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableImportAddressFilter`

If set this enables the Import Address Filter mitigation in enforcement mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditImportAddressFilter`

If set this enables the Import Address Filter mitigation in enforcement mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableRopStackPivot`

If set this enables the stack pivot anti-ROP (Return-oriented-programming) mitigation in enforcement mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditRopStackPivot`

If set this enables the stack pivot anti-ROP (Return-oriented-programming) mitigation in audit mode for the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableRopCallerCheck`

If set this enables the caller check anti-ROP (Return-oriented-programming) mitigation in enforcement mode for the process. Applies to 32-bit processes only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditRopCallerCheck`

If set this enables the caller check anti-ROP (Return-oriented-programming) mitigation in audit mode for the process. Applies to 32-bit processes only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableRopSimExec`

If set this enables the simulated execution anti-ROP (Return-oriented-programming) mitigation in enforcement mode for the process. Applies to 32-bit processes only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditRopSimExec`

If set this enables the simulated execution anti-ROP (Return-oriented-programming) mitigation in audit mode for the process. Applies to 32-bit processes only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ntddk.h

PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY structure

1/30/2019 • 2 minutes to read • Edit Online

This structure is not supported.

Syntax

```
typedef struct _PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY {
  union {
    ULONG Flags;
    struct {
      ULONG FilterId : 4;
      ULONG ReservedFlags : 28;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY, *PPROCESS_MITIGATION_SYSTEM_CALL_FILTER_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.FilterId`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ntddk.h (include Ntddk.h)

PS_CREATE_NOTIFY_INFO structure

2/5/2019 • 2 minutes to read • Edit Online

The **PS_CREATE_NOTIFY_INFO** structure provides information about a newly created process.

Syntax

```
typedef struct _PS_CREATE_NOTIFY_INFO {
  SIZE_T              Size;
  union {
    ULONG Flags;
    struct {
      ULONG FileOpenNameAvailable : 1;
      ULONG IsSubsystemProcess : 1;
      ULONG Reserved : 30;
    };
  };
  HANDLE              ParentProcessId;
  CLIENT_ID           CreatingThreadId;
  struct _FILE_OBJECT *FileObject;
  PCUNICODE_STRING    ImageFileName;
  PCUNICODE_STRING    CommandLine;
  NTSTATUS            CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

Members

`Size`

The size, in bytes, of this structure. The operating system uses this size to indicate the type of structure that it passes to CreateProcessNotifyEx. Currently, this member is always **sizeof**(**PS_CREATE_NOTIFY_INFO**).

`Flags`

Reserved. Use the **FileOpenNameAvailable** member instead.

`FileOpenNameAvailable`

A Boolean value that specifies whether the **ImageFileName** member contains the exact file name that is used to open the process executable file.

`IsSubsystemProcess`

A Boolean value that indicates the type of process subsystem is a subsystem other than Win32.

> **Note**
>
> **IsSubsystemProcess** is only populated for subsystem processes other than Win32 when a driver has registered through PsSetCreateProcessNotifyRoutineEx2 with a type that allows for notifications from subsystem processes. When **IsSubsystemProcess** is set, the **FileObject**, **ImageFileName**, and **CommandLine** may be NULL. Drivers should use **ProcessSubsystemInformation** to query the subsystem type if needed. For more information, see NtQueryInformationProcess.

`Reserved`

Reserved for system use.

`ParentProcessId`

The process ID of the parent process for the new process. Note that the parent process is not necessarily the same process as the process that created the new process. The new process can inherit certain properties of the parent process, such as handles or shared memory. (The process ID of the process creator is given by **CreatingThreadId->UniqueProcess**.)

`CreatingThreadId`

The process ID and thread ID of the process and thread that created the new process. **CreatingThreadId->UniqueProcess** contains the process ID, and **CreatingThreadId->UniqueThread** contains the thread ID.

`FileObject`

A pointer to the file object for the process executable file.

> **Note**
> If **IsSubsystemProcess** is TRUE, this value may be NULL.

`ImageFileName`

A pointer to a UNICODE_STRING string that holds the file name of the executable. If the **FileOpenNameAvailable** member is **TRUE**, the string specifies the exact file name that is used to open the executable file. If **FileOpenNameAvailable** is **FALSE**, the operating system might provide only a partial name.

> **Note**
> If **IsSubsystemProcess** is TRUE, this value maybe NULL.

`CommandLine`

A pointer to a **UNICODE_STRING** string that holds the command that is used to execute the process. If the command is not available, **CommandLine** is **NULL**.

> **Note**
> If **IsSubsystemProcess** is TRUE, this value maybe NULL.

`CreationStatus`

The NTSTATUS value to return for the process-creation operation. Drivers can change this value to an error code to prevent the process from being created.

Requirements

Minimum supported client	Available in Windows Vista and later versions of the Windows operating system.
Header	ntddk.h (include Ntddk.h)

See Also

CreateProcessNotifyEx

PsSetCreateProcessNotifyRoutineEx

UNICODE_STRING

PSCREATEPROCESSNOTIFYTYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Indicates the type of process notification. This enumeration is used in PsSetCreateProcessNotifyRoutineEx2 to register callback notifications.

Syntax

```
typedef enum _PSCREATEPROCESSNOTIFYTYPE {
  PsCreateProcessNotifySubsystems
} PSCREATEPROCESSNOTIFYTYPE;
```

Constants

PsCreateProcessNotifySubsystems	Indicates that the driver-registered callback is invoked for processes of all subsystems, including the Win32 subsystem. Drivers can call NtQueryInformationProcess to determine the underlying subsystem. The query retrieves a SUBSYSTEM_INFORMATION_TYPE value.

Requirements

Minimum supported client	Windows 10, version 1703
Minimum supported server	Windows Server 2016
Header	ntddk.h (include Ntddk.h)

See Also

NtQueryInformationProcess

PsSetCreateProcessNotifyRoutineEx2

SUBSYSTEM_INFORMATION_TYPE

PSCREATETHREADNOTIFYTYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Indicates the type of thread notification. This enumeration is used in PsSetCreateThreadNotifyRoutineEx to register callback notifications associated with thread creation or deletion.

Syntax

```
typedef enum _PSCREATETHREADNOTIFYTYPE {
  PsCreateThreadNotifyNonSystem,
  PsCreateThreadNotifySubsystems
} PSCREATETHREADNOTIFYTYPE;
```

Constants

PsCreateThreadNotifyNonSystem	The driver-registered callback function is executed on the new non-system thread, which enables the callback function to perform tasks such as setting the initial thread context.
PsCreateThreadNotifySubsystems	Indicates that the driver-registered callback function is invoked for threads of all subsystems. Drivers can call NtQueryInformationThread to determine the underlying subsystem. The query retrieves a SUBSYSTEM_INFORMATION_TYPE value.

Requirements

Minimum supported client	Windows 10, version 1703
Minimum supported server	Windows Server 2016
Header	ntddk.h (include Ntddk.h)

See Also

NtQueryInformationThread

PsSetCreateThreadNotifyRoutineEx

SUBSYSTEM_INFORMATION_TYPE

SILO_MONITOR_REGISTRATION structure

2/5/2019 • 2 minutes to read • Edit Online

This structure specifies a server silo monitor that can receive notifications about server silo events.

Syntax

```
typedef struct _SILO_MONITOR_REGISTRATION {
  UCHAR                          Version;
  BOOLEAN                        MonitorHost;
  BOOLEAN                        MonitorExistingSilos;
  UCHAR                          Reserved[5];
  union {
    PUNICODE_STRING DriverObjectName;
    PUNICODE_STRING ComponentName;
  };
  SILO_MONITOR_CREATE_CALLBACK    CreateCallback;
  SILO_MONITOR_TERMINATE_CALLBACK TerminateCallback;
} SILO_MONITOR_REGISTRATION, *PSILO_MONITOR_REGISTRATION;
```

Members

`Version`

Set to **SERVER_SILO_MONITOR_REGISTRATION_VERSION**.

`MonitorHost`

If **true**, a create notification will be delivered for the host context.

`MonitorExistingSilos`

If **true**, create and terminate notifications will be delivered for any silos that currently exist at the time of registration; otherwise, only notifications for new silos will be delivered.

`Reserved`

Reserved for system use.

`DriverObjectName`

`ComponentName`

`CreateCallback`

A pointer to a callback that is invoked whenever a new server silo is created on the system. This value may be **NULL**. This gives drivers to opportunity to handle the event and set up per-silo data structures.

`TerminateCallback`

A pointer to a callback that is invoked whenever a server silo is terminated (about to be destroyed) on the system. This value may be **NULL**. This gives drivers the opportunity to complete work within the silo and begin tearing down their per-silo data structures.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Header	ntddk.h

SUBSYSTEM_INFORMATION_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Indicates the type of subsystem for a process or thread. This enumeration is used in NtQueryInformationProcess and NtQueryInformationThread calls.

Syntax

```
typedef enum _SUBSYSTEM_INFORMATION_TYPE {
  SubsystemInformationTypeWin32,
  SubsystemInformationTypeWSL,
  MaxSubsystemInformationType
} SUBSYSTEM_INFORMATION_TYPE, *PSUBSYSTEM_INFORMATION_TYPE;
```

Constants

SubsystemInformationTypeWin32	The subsystem type for the process or thread is Win32.
SubsystemInformationTypeWSL	The subsystem type for the process or thread is Windows Subsystem for Linux (WSL). For this process, these members of the PS_CREATE_NOTIFY_INFO structure are set as follows: • The **FileObject** member is the NTFS file object from LxFs or DriveFs, the file system that is used for interoperability with Windows. • The **ImageFileName** member is the Linux path of the image file. • The **CommandLine** member is the Linux NULL-separated command line. The preceding member values may be NULL.
MaxSubsystemInformationType	Reserved.

Requirements

Header	ntddk.h

See Also

NtQueryInformationProcess

NtQueryInformationThread

CORRELATION_VECTOR structure

1/25/2019 • 2 minutes to read • Edit Online

Store the correlation vector that is used to reference events and the generated logs for diagnostic purposes.

Syntax

```
typedef struct CORRELATION_VECTOR {
  CHAR Version;
  CHAR Vector[RTL_CORRELATION_VECTOR_STRING_LENGTH];
} CORRELATION_VECTOR;
```

Members

`Version`

The version of the correlation vector. Possible values are:

- RTL_CORRELATION_VECTOR_VERSION_1
- RTL_CORRELATION_VECTOR_VERSION_2
- RTL_CORRELATION_VECTOR_VERSION_CURRENT

`Vector`

An array CHARs that represents the correlation vector.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ntddk.h

DRIVER_REINITIALIZE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *Reinitialize* routine continues driver and device initialization after the driver's **DriverEntry** routine returns.

Syntax

```
DRIVER_REINITIALIZE DriverReinitialize;

void DriverReinitialize(
  _DRIVER_OBJECT *DriverObject,
  PVOID Context,
  ULONG Count
)
{...}
```

Parameters

`*DriverObject`

Caller-supplied pointer to a DRIVER_OBJECT structure. This is the driver's driver object.

`Context`

Caller-supplied pointer to context information, specified in a previous call to IoRegisterDriverReinitialization or IoRegisterBootDriverReinitialization.

`Count`

Caller-supplied value representing the number of times the *Reinitialize* routine has been called, including the current call.

Return Value

None

Remarks

To queue a *Reinitialize* routine for execution, a driver's DriverEntry routine must call either IoRegisterDriverReinitialization or IoRegisterBootDriverReinitialization. The *Reinitialize* routine can also call **IoRegisterDriverReinitialization** itself, which causes the routine to be requeued. This requeuing can occur multiple times, and the routine's *Count* parameter indicates the number of times it has been called. The first call to **IoRegisterDriverReinitialization** must be made from **DriverEntry**, and **DriverEntry** must return STATUS_SUCCESS.

For more information about implementing a *Reinitialize* routine, see Writing a Reinitialize Routine.

Examples

To define a *Reinitialize* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *Reinitialize* callback routine that is named `MyReinitialize`, use the DRIVER_REINITIALIZE type as shown in this code example:

```
DRIVER_REINITIALIZE MyReinitialize;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyReinitialize(
    struct _DRIVER_OBJECT  *DriverObject,
    PVOID  Context,
    ULONG  Count
    )
  {
      // Function body
  }
```

The DRIVER_REINITIALIZE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_REINITIALIZE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

ExFreePool macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreePool** routine deallocates a block of pool memory.

Syntax

```
void ExFreePool(
   a
);
```

Parameters

`a`

Specifies the address of the block of pool memory being deallocated.

Return Value

None

Remarks

This routine releases memory allocated by ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePoolWithQuota, or ExAllocatePoolWithQuotaTag. The memory block must not be accessed after it is freed.

Drivers can also use the **ExFreePoolWithTag** routine to free buffers allocated by ExAllocatePoolWithTag and ExAllocatePoolWithQuotaTag.

Callers of **ExFreePool** must be running at IRQL <= DISPATCH_LEVEL. A caller at DISPATCH_LEVEL must have specified a **NonPaged***XxxPoolType* when the memory was allocated. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

DDI compliance rules	DoubleExFreePool

See Also

ExAllocatePool

ExAllocatePoolWithQuota

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExFreePoolWithTag

EXPAND_STACK_CALLOUT callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ExpandedStackCall* routine executes with a guaranteed stack size.

Syntax

```
EXPAND_STACK_CALLOUT ExpandStackCallout;

VOID() ExpandStackCallout(
  PVOID Parameter
)
{...}
```

Parameters

`Parameter`

The value passed to the KeExpandKernelStackAndCallout routine that executed *ExpandedStackCall*.

Return Value

None

Remarks

The *ExpandedStackCall* routine must handle all exceptions. Any unhandled exception causes the system to bug check with Bug Check 0x1E: KMODE_EXCEPTION_NOT_HANDLED.

If the *ExpandedStackCall* changes the current IRQL, it must restore the original value before returning.

Examples

To define an *ExpandedStackCall* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *ExpandedStackCall* callback routine that is named `MyExpandedStackCall`, use the EXPAND_STACK_CALLOUT type as shown in this code example:

```
EXPAND_STACK_CALLOUT MyExpandedStackCall;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyExpandedStackCall(
    PVOID  Parameter
    )
  {
      // Function body
  }
```

The EXPAND_STACK_CALLOUT function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the EXPAND_STACK_CALLOUT function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Drivers can implement ExpandedStackCall routines on Windows Server 2003 for an x64-based processor, and on Windows Vista and later versions of Windows for all processors.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)

See Also

KeExpandKernelStackAndCallout

ExRaiseAccessViolation function

1/25/2019 • 2 minutes to read • Edit Online

The **ExRaiseAccessViolation** routine can be used with structured exception handling to throw a driver-determined exception for a memory access violation that occurs when a driver processes I/O requests.

Syntax

```
void ExRaiseAccessViolation(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

ExRaiseAccessViolation raises an exception with the exception code set to STATUS_ACCESS_VIOLATION.

Because **ExRaiseAccessViolation** can only be used at IRQL = PASSIVE_LEVEL, only high-level drivers typically use this routine—for example, file system drivers.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	IrqlExPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ExRaiseDatatypeMisalignment

ExRaiseStatus

IoAllocateErrorLogEntry

KeBugCheckEx

ExRaiseDatatypeMisalignment function

1/25/2019 • 2 minutes to read • Edit Online

The **ExRaiseDatatypeMisalignment** routine can be used with structured exception handling to throw a driver-determined exception for a misaligned data type that occurs when a driver processes I/O requests.

Syntax

```
void ExRaiseDatatypeMisalignment(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

ExRaiseDatatypeMisalignment raises an exception with the exception code set to STATUS_DATATYPE_MISALIGNMENT.

Because **ExRaiseDatatypeMisalignment** can only be used at IRQL = PASSIVE_LEVEL, only high-level drivers typically use this routine — for example, file system drivers.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	IrqlExPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ExRaiseAccessViolation

ExRaiseStatus

IoAllocateErrorLogEntry

KeBugCheckEx

ExUuidCreate function

1/25/2019 • 2 minutes to read • Edit Online

The **ExUuidCreate** routine initializes a UUID (GUID) structure to a newly generated value.

Syntax

```
NTSTATUS ExUuidCreate(
  UUID *Uuid
);
```

Parameters

`Uuid`

A pointer to a caller-allocated UUID (GUID) structure that is set to a new UUID value.

Return Value

Possible return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully generated a UUID that is universally unique.
RPC_NT_UUID_LOCAL_ONLY	The routine generated a UUID that is unique only to this computer. This can occur when the MAC address is not an IEEE universally-administered address or when no NICs are present.
STATUS_RETRY	The system is not ready to generate a new UUID.

Remarks

A UUID and a GUID are the same data type.

The caller can iteratively attempt to obtain a new UUID value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlExPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

HalAllocateHardwareCounters function

1/25/2019 • 2 minutes to read • Edit Online

The **HalAllocateHardwareCounters** routine allocates a set of hardware performance counters.

Syntax

```
NTHALAPI NTSTATUS HalAllocateHardwareCounters(
  PGROUP_AFFINITY                GroupAffinty,
  ULONG                          GroupCount,
  PPHYSICAL_COUNTER_RESOURCE_LIST ResourceList,
  PHANDLE                        CounterSetHandle
);
```

Parameters

`GroupAffinty`

Reserved for future use. Set this parameter to **NULL**.

`GroupCount`

Reserved for future use. Set this parameter to zero.

`ResourceList`

Reserved for future use. Set this parameter to **NULL**.

`CounterSetHandle`

A pointer to a location into which the routine writes a handle to the allocated counter resources. To release these resources later, the caller must pass this handle to the HalFreeHardwareCounters routine. If the requested counter resources are unavailable, **HalAllocateHardwareCounters** sets **CounterSetHandle* = **NULL** and returns STATUS_INSUFFICIENT_RESOURCES.

Return Value

HalAllocateHardwareCounters returns STATUS_SUCCESS if the call was successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_INSUFFICIENT_RESOURCES	The requested counter resources are currently unavailable.

STATUS_INVALID_PARAMETER	The caller specified an invalid parameter value.

Remarks

Most processors have performance monitor units (PMUs) that contain a number of hardware counters. Software tools use these counters to monitor various aspects of system performance. Typically, such a tool consists of a custom kernel-mode driver to program the counters and a user-mode application that communicates with the driver.

If more than one such tool is installed on a computer, the associated drivers must avoid trying to use the same hardware counters simultaneously. To avoid such resource conflicts, all drivers that use counter resources should use the **HalAllocateHardwareCounters** and **HalFreeHardwareCounters** routines to coordinate their sharing of these resources.

A counter resource is a single hardware counter, a block of contiguous counters, or a counter overflow interrupt in a PMU.

Before configuring the counters, a driver can call the **HalAllocateHardwareCounters** routine to acquire exclusive access to a set of counter resources. After the driver no longer needs these resources, it must free the resources by calling the **HalFreeHardwareCounters** routine.

In Windows 8 and Windows 7, a successful call to **HalAllocateHardwareCounters** grants the caller exclusive access to all counter resources in the performance monitor unit of a single-processor system. In a multiprocessor system, a successful call grants the caller exclusive access to all counter resources in all processors in the system. Future versions of this routine might provide more fine-grained allocation of counter resources.

Virtualization software typically does not virtualize hardware performance counters. Thus, these counters might not be available in a virtual machine, regardless of whether **HalAllocateHardwareCounters** returns a status code of STATUS_SUCCESS. For example, hardware performance counters are not available in a Hyper-V virtual machine, but **HalAllocateHardwareCounters** might still return STATUS_SUCCESS.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	Hal.lib
DLL	Hal.dll

IRQL	PASSIVE_LEVEL

See Also

GROUP_AFFINITY

HalFreeHardwareCounters

PHYSICAL_COUNTER_RESOURCE_LIST

HalExamineMBR function

1/25/2019 • 2 minutes to read • Edit Online

The **HalExamineMBR** routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.

Syntax

```
void HalExamineMBR(
  PDEVICE_OBJECT DeviceObject,
  ULONG          SectorSize,
  ULONG          MBRTypeIdentifier,
  PVOID          *Buffer
);
```

Parameters

`DeviceObject`

A pointer to the device object for the device being examined.

`SectorSize`

The minimum number of bytes that an I/O operation can fetch from the device being examined. If this value is less than 512, **HalExamineMBR** reads 512 bytes to ensure that it reads an entire partition table.

`MBRTypeIdentifier`

MBR partition type identifier. This parameter specifies the type of MBR that may be on the disk. For more information, see Remarks.

`Buffer`

A pointer to a location to which **HalExamineMBR** writes a pointer to a buffer that contains data from the MBR. The layout of the buffer depends on the MBR partition type. **HalExamineMBR** allocates the storage for this buffer. The caller must deallocate this buffer as soon as possible by calling the ExFreePool routine.

HalExamineMBR sets **Buffer* = **NULL** if the MBR partition type of the disk does not match that specified by *MBRTypeIdentifier* or if there is an error.

Return Value

None

Remarks

For a list of system-defined MBR partition type identifiers, see the table in PARTITION_INFORMATION. These identifiers are defined in the Ntdddisk.h header file.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ExFreePool

PARTITION_INFORMATION

HalFreeHardwareCounters function

1/25/2019 • 2 minutes to read • Edit Online

The **HalFreeHardwareCounters** routine frees a set of hardware performance counters that was acquired in a previous call to HalAllocateHardwareCounters routine.

Syntax

```
NTSTATUS HalFreeHardwareCounters(
  _In_ HANDLE CounterSetHandle
);
```

Parameters

`CounterSetHandle`

A handle to the allocated counter resources. The caller acquired this handle in a previous call to **HalAllocateHardwareCounters**.

Return Value

HalFreeHardwareCounters returns STATUS_SUCCESS if the call was successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ PAR AM ETE R	Parameter *CounterSetHandle* is not a valid counter resources handle.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	Hal.lib

DLL	Hal.dll
IRQL	PASSIVE_LEVEL

See Also

HalAllocateHardwareCounters

IoAllocateController function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateController** routine sets up the call to a driver-supplied ControllerControl routine as soon as the device controller, represented by the given controller object, is available to carry out an I/O operation for the target device, represented by the given device object.

Syntax

```
void IoAllocateController(
  PCONTROLLER_OBJECT ControllerObject,
  PDEVICE_OBJECT     DeviceObject,
  PDRIVER_CONTROL    ExecutionRoutine,
  PVOID              Context
);
```

Parameters

`ControllerObject`

Pointer to a driver-created controller object, usually representing a physical controller to be allocated for an I/O operation on an attached device.

`DeviceObject`

Pointer to the device object, representing the target device of the current IRP.

`ExecutionRoutine`

Pointer to the driver-supplied ControllerControl routine.

`Context`

Pointer to a driver-determined context, passed to the driver's *ControllerControl* routine when it is called.

Return Value

None

Remarks

This routine reserves exclusive access to the hardware controller for the specified device.

The ControllerControl routine returns a value indicating whether the controller remains allocated to the device, either **DeallocateObject** or **KeepObject**. If it returns **KeepObject**, the driver must subsequently call **IoFreeController** to release the controller object.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, HwStorPortProhibitedDDIs, IrqlDispatch(storport), SpNoWait, StorPortStartIo

See Also

ControllerControl

IoCreateController

IoDeleteController

IoFreeController

IoAssignArcName macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoAssignArcName** routine creates a symbolic link between the ARC name of a physical device and the name of the corresponding device object when it has been created.

Syntax

```
void IoAssignArcName(
   ArcName,
   DeviceName
);
```

Parameters

`ArcName`

Pointer to a buffer containing the ARC name of the device. The ARC name must be a Unicode string.

`DeviceName`

Pointer to a buffer containing the name of the device object, representing the same device. The device object name must be a Unicode string.

Return Value

None

Remarks

Drivers of hard disk devices need not call this routine. Drivers of other mass storage devices, including floppy, CD-ROM, and tape devices, should call **IoAssignArcName** during their initialization.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoCreateDevice

IoClearActivityIdThread function

1/25/2019 • 2 minutes to read • Edit Online

The IoClearActivityIdThread routine clears the activity ID of the current thread.

Syntax

```
void IoClearActivityIdThread(
  LPCGUID OriginalId
);
```

Parameters

`OriginalId`

The activity ID that was previously set on the thread.

Return Value

None.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

IoCreateController function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCreateController** routine allocates memory for and initializes a controller object with a controller extension of a driver-determined size.

Syntax

```
PCONTROLLER_OBJECT IoCreateController(
  ULONG Size
);
```

Parameters

`Size`

Specifies the number of bytes to be allocated for the controller extension.

Return Value

IoCreateController returns a pointer to the controller object or a **NULL** pointer if memory could not be allocated for the requested device extension.

Remarks

A controller object usually represents a physical device controller with attached devices on which a single driver carries out I/O requests. The controller extension is allocated from nonpaged pool and is guaranteed to be accessible by any driver routine and in an arbitrary thread context.

The controller object is used to synchronize I/O operations to target devices for which I/O requests can come in concurrently to a single, monolithic driver. A driver also might use a controller object to synchronize operations through device channels.

If **IoCreateController** returns **NULL**, the driver should fail device start-up.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive2, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

CONTROLLER_OBJECT

IoAllocateController

IoDeleteController

IoFreeController

IoDeassignArcName macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoDeassignArcName** routine removes a symbolic link between the ARC name for a device and the named device object.

Syntax

```
void IoDeassignArcName(
   ArcName
);
```

Parameters

`ArcName`

Pointer to a buffered Unicode string that is the ARC name.

Return Value

None

Remarks

IoDeassignArcName is generally called if the driver is deleting the device object, for example, when the driver is unloading.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoAssignArcName

IoDecrementKeepAliveCount function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDecrementKeepAliveCount** routine decrements a reference count associated with an Windows app on a specific device. This routine is called by a kernel mode driver when the app's I/O process has completed. If the app's reference count drops to 0, Windows can suspend the app.

Syntax

```
NTSTATUS IoDecrementKeepAliveCount(
  PFILE_OBJECT FileObject,
  PEPROCESS    Process
);
```

Parameters

`FileObject`

The file object handle to the device.

`Process`

The process associated with the device.

Return Value

This routine returns **STATUS_SUCCESS** on success, or the appropriate **NTSTATUS** error code on failure. **NTSTATUS** error codes are defined in Ntstatus.h.

Requirements

Minimum supported client	Available in Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe

IoDeleteController function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDeleteController** routine removes a given controller object from the system, for example, when the driver that created it is being unloaded.

Syntax

```
void IoDeleteController(
  PCONTROLLER_OBJECT ControllerObject
);
```

Parameters

`ControllerObject`

Pointer to the controller object to be released.

Return Value

None

Remarks

IoDeleteController deallocates the memory for the controller object, including the controller extension.

This routine must be called when a driver that created a controller object is being unloaded or when the driver encounters a fatal error during device start-up, such as being unable to properly initialize a physical device.

A driver must release certain resources for which the driver supplied storage in its controller extension before it calls **IoDeleteController**. For example, if the driver stores the pointer to its interrupt object(s) in the controller extension, it must call IoDisconnectInterrupt before **IoDeleteController**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoCreateController

IoDisconnectInterrupt

IoFreeController function

1/25/2019 • 2 minutes to read • Edit Online

The **IoFreeController** routine releases a previously allocated controller object when the driver has completed an I/O request.

Syntax

```
void IoFreeController(
  PCONTROLLER_OBJECT ControllerObject
);
```

Parameters

`ControllerObject`

Pointer to the driver's controller object, which was allocated for the current I/O operation on a particular device by calling **IoAllocateController**.

Return Value

None

Remarks

The connection between the current target device object and the controller object is released only if no requests are currently queued to the same device. Otherwise, the driver's ControllerControl routine is called with the next IRP bound through the device controller to the target device.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, HwStorPortProhibitedDDIs, IrqlDispatch(storport)

See Also

IoAllocateController

IoCreateController

IoDeleteController

IoGetActivityIdIrp function

1/25/2019 • 2 minutes to read • Edit Online

The IoGetActivityIdIrp routine retrieves the current activity ID associated with an IRP.

Syntax

```
NTSTATUS IoGetActivityIdIrp(
  PIRP   Irp,
  LPGUID Guid
);
```

Parameters

`Irp`

The IRP from which to retrieve the activity ID.

`Guid`

A pointer to a location to store the retrieved GUID.

Return Value

IoGetActivityIdIrp returns STATUS_SUCCESS if the call is successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STA TUS _NO T_F OU ND	No activity ID is associated with the request.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level

IoGetActivityIdThread function

1/25/2019 • 2 minutes to read • Edit Online

The IoGetActivityIdThread routine returns the activity ID associated with the current thread.

Syntax

```
LPCGUID IoGetActivityIdThread(

);
```

Parameters

This function has no parameters.

Return Value

The activity ID associated with the current thread.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

IoGetConfigurationInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetConfigurationInformation** routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.

Syntax

```
PCONFIGURATION_INFORMATION IoGetConfigurationInformation(

);
```

Parameters

This function has no parameters.

Return Value

IoGetConfigurationInformation returns a pointer to the configuration information structure. This structure is defined as follows:

```
typedef struct _CONFIGURATION_INFORMATION {

    //
    // This field indicates the total number of disks in the system. This
    // number should be used by the driver to determine the names of new
    // disks. This field should be updated by the driver as it finds new
    // disks.
    //

    ULONG DiskCount;                // Count of hard disks thus far
    ULONG FloppyCount;              // Count of floppy disks thus far
    ULONG CdRomCount;               // Count of CD-ROM drives thus far
    ULONG TapeCount;                // Count of tape drives thus far
    ULONG ScsiPortCount;            // Count of SCSI port adapters thus far
    ULONG SerialCount;              // Count of serial devices thus far
    ULONG ParallelCount;            // Count of parallel devices thus far

    //
    // These next two fields indicate ownership of the two I/O address
    // spaces that are used by WD1003-compatible disk controllers.
    //

    BOOLEAN AtDiskPrimaryAddressClaimed;    // 0x1F0 - 0x1FF
    BOOLEAN AtDiskSecondaryAddressClaimed;  // 0x170 - 0x17F

    //
    // Indicates the structure version, as anything value beyond this will have been added.
    // Use the structure size as the version.
    //

    ULONG Version;

    //
    // Indicates the total number of medium changer devices in the system.
    // This field will be updated by the drivers as it determines that
    // new devices have been found and will be supported.
    //

    ULONG MediumChangerCount;

} CONFIGURATION_INFORMATION, *PCONFIGURATION_INFORMATION;
```

Remarks

Certain types of device drivers can use the configuration information structure's values to construct device object names with appropriate digit suffixes when each driver creates its device objects. Note that the digit suffix for device object names is a zero-based count, while the counts maintained in the configuration information structure represent the number of device objects of a particular type already created. That is, the configuration information counts are one-based.

Any driver that calls **IoGetConfigurationInformation** must increment the count for its kind of device in this structure when it creates a device object to represent a physical device.

The system-supplied SCSI port driver supplies the count of SCSI HBAs present in the computer. SCSI class drivers can read this value to determine how many HBA-specific miniport drivers might control a SCSI bus with an attached device of the class driver's type.

The configuration information structure also contains a value indicating whether an already loaded driver has claimed either of the "AT" disk I/O address ranges.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

IoAssignResources

IoQueryDeviceDescription

IoReportResourceUsage

IoGetFileObjectGenericMapping function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetFileObjectGenericMapping** routine returns information about the mapping between each generic access right and the set of specific access rights for file objects.

Syntax

```
PGENERIC_MAPPING IoGetFileObjectGenericMapping(

);
```

Parameters

This function has no parameters.

Return Value

IoGetFileObjectGenericMapping returns a pointer to the GENERIC_MAPPING structure that describes the mapping between each generic access right and the implied set of specific access rights.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

GENERIC_MAPPING

IoCheckShareAccess

IoSetShareAccess

ZwCreateFile

IoGetInitiatorProcess function

1/25/2019 • 2 minutes to read • Edit Online

The IoGetInitiatorProcess routine retrieves the process which initiated the creation of a file object if different than the process which is issuing the create.

Syntax

```
PEPROCESS IoGetInitiatorProcess(
  PFILE_OBJECT FileObject
);
```

Parameters

`FileObject`

The file object from which to retrieve the initiator process.

Return Value

NULL if there is no initiator process. Otherwise, a pointer to which process initiated the creation of the file object.

Remarks

A driver normally uses IoGetInitiatorProcess to determine which process has issued a request. However, there are situations where a system component may issue a create on behalf of another process (after a successful create the component will duplicate the handle to the process). This routine can be used if the driver must know which process the create operation is ultimately intended for.

Requirements

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

IoGetPagingIoPriority function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetPagingIoPriority** routine indicates the priority level of a paging I/O request.

Syntax

```
IO_PAGING_PRIORITY IoGetPagingIoPriority(
  PIRP Irp
);
```

Parameters

`Irp`

Pointer to the IRP to be tested for paging priority.

Return Value

IoGetPagingIoPriority returns the IO_PAGING_PRIORITY value for the associated IRP.

Remarks

For I/O requests that causing paging, the system associates an **IO_PAGING_PRIORITY** value that describes the IRP's priority. Drivers can use this value when queuing IRPs.

Requirements

Minimum supported client	Available in Microsoft Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_PAGING_PRIORITY

IoIncrementKeepAliveCount function

1/25/2019 • 2 minutes to read • Edit Online

The **IoIncrementKeepAliveCount** routine increments a reference count associated with an Windows app process on a specific device. This routine is called by a kernel mode driver in response to the app opening a process for I/O. This prevents Windows from suspending the app before the I/O process is complete.

Syntax

```
NTSTATUS IoIncrementKeepAliveCount(
  PFILE_OBJECT FileObject,
  PEPROCESS    Process
);
```

Parameters

`FileObject`

The file object handle to the device.

`Process`

The process associated with the device.

Return Value

This routine returns **STATUS_SUCCESS** on success, or the appropriate **NTSTATUS** error code on failure. **NTSTATUS** error codes are defined in Ntstatus.h.

Requirements

Minimum supported client	Available in Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe

See Also

IoDecrementKeepAliveCount

IoIsValidIrpStatus function

1/25/2019 • 2 minutes to read • Edit Online

The **IoIsValidIrpStatus** routine validates the specified NTSTATUS status code value.

Syntax

```
LOGICAL IoIsValidIrpStatus(
  IN NTSTATUS Status
);
```

Parameters

`Status`

The status code to be validated.

Return Value

IoIsValidIrpStatus returns **TRUE** if the *Status* parameter is a well-formed NTSTATUS value. Otherwise, the routine returns **FALSE**.

Remarks

Driver developers can define custom NTSTATUS values for use as status codes for IRP_MJ_INTERNAL_DEVICE_CONTROL requests. Drivers can call **IoIsValidIrpStatus** to validate the status codes for these requests. This routine is useful for detecting an uninitialized or corrupted **Status** value in the I/O status block of an IRP.

IoIsValidIrpStatus detects invalid status codes that obviously violate the rules for defining custom NTSTATUS values, but might not detect invalid status codes that appear to follow the rules. The routine might return **TRUE** for a *Status* parameter that is not a valid NTSTATUS code value, but rarely, if ever, returns **FALSE** for a valid NTSTATUS code.

IoIsValidIrpStatus examines the **Sev**, **Facility**, and **Code** fields in the *Status* parameter value to determine whether the combination of values in these fields conforms to the published guidelines. If the **C** bit of the *Status* parameter is set, **IoIsValidIrpStatus** returns **TRUE** regardless of the other field values in *Status*.

For more information about these fields, see Defining New NTSTATUS Values.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level.

See Also

IRP_MJ_INTERNAL_DEVICE_CONTROL

IoMakeAssociatedIrp function

1/25/2019 • 2 minutes to read • Edit Online

This routine is reserved for use by file systems and file system filter drivers. The **IoMakeAssociatedIrp** routine allocates and initializes an IRP to be associated with a master IRP sent to a highest-level driver, allowing the caller to split the original request and send associated IRPs on to lower-level drivers.

Syntax

```
PIRP IoMakeAssociatedIrp(
  PIRP  Irp,
  CCHAR StackSize
);
```

Parameters

`Irp`

Pointer to the master IRP that was input to a highest-level driver's Dispatch routine.

`StackSize`

Specifies the number of stack locations to be allocated for the associated IRP. The value must be at least equal to the *StackSize* of the next-lower driver's device object, but the associated IRP can have an additional stack location for the caller.

Return Value

IoMakeAssociatedIrp returns a pointer to the associated IRP or returns a **NULL** pointer if an IRP cannot be allocated.

Remarks

Only a highest-level driver can call this routine.

The I/O manager completes the master IRP automatically when lower drivers have completed all associated IRPs as long as the caller has not set its IoCompletion routine in an associated IRP and returned STATUS_MORE_PROCESSING_REQUIRED from its *IoCompletion* routine. In these circumstances, the caller must explicitly complete the master IRP when that driver has determined that all associated IRPs were completed.

Only the master IRP is associated with a thread; associated IRPs are not. For this reason, the I/O manager cannot call Cancel routines for associated IRPs when a thread exits. When the master IRP's thread exits, the I/O manager calls the master IRP's *Cancel* routine. The *Cancel* routine is responsible for tracking down all associated IRPs and calling IoCancelIrp to cancel them.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IRP

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoCallDriver

IoSetCompletionRoutine

IoPropagateActivityIdToThread function

1/25/2019 • 2 minutes to read • Edit Online

The **IoPropagateActivityIdToThread** routine associates the activity ID from an IRP with the current thread.

Syntax

```
NTSTATUS IoPropagateActivityIdToThread(
  PIRP    Irp,
  LPGUID  PropagatedId,
  LPCGUID *OriginalId
);
```

Parameters

`Irp`

The IRP whose ID will be propagated to the thread.

`PropagatedId`

A pointer to memory allocated by the caller to store the ID in the thread.

`OriginalId`

Upon successfully returning from the call, holds the ID that was previously set on the thread. The driver must call **IoClearActivityIdThread** with this pointer when tracing is completed within the same thread context.

Return Value

IoPropagateActivityIdToThread returns STATUS_SUCCESS if the call is successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STA TUS _NO T_F OU ND	The IRP does not have an ID associated with it.

Remarks

This routine should be used by drivers that are tracing aware and are issuing I/O on a worker thread. Note that such drivers must call IoClearActivityIdThread with the *OriginalId* before they return control from the thread, if the call was successful.

Drivers that use I/O work items do not need to call this routine because the I/O subsystem takes care of propagating activity IDs to threads in that case.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

IoQueryFullDriverPath function

1/25/2019 • 2 minutes to read • Edit Online

The **IoQueryFullDriverPath** routine retrieves the full path name of the binary file that is loaded for the specified driver object. Starting in Windows 10 version 1709, callers may query for driver objects that are not their own, as long as they use proper synchronization to ensure that the DRIVER_OBJECT structure remains valid during the call.

Syntax

```
NTSTATUS IoQueryFullDriverPath(
  PDRIVER_OBJECT  DriverObject,
  PUNICODE_STRING FullPath
);
```

Parameters

`DriverObject`

A pointer to a DRIVER_OBJECT structure. If you are calling **IoQueryFullDriverPath** on a computer running a version of Windows 10 earlier than version 1709, this structure is required to be the driver object for the calling driver.

`FullPath`

A pointer to a caller-allocated UNICODE_STRING structure. On successful return, this structure contains the path name.

Return Value

IoQueryFullDriverPath returns STATUS_SUCCESS if the call successfully fetches the path name. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_ACCESS_DENIED	The target driver object does not belong to the caller. This status code is only returned on versions of Windows 10 earlier than 1709.
STATUS_NOT_FOUND	The driver object has no section (loaded memory image) associated with it.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	Insufficient resources are available to perform the requested operation.

Remarks

A driver can call this routine to query for the full path name of its binary file, or, starting in Windows 10 version 1709, the full path name of the binary file for another driver.

The caller allocates the UNICODE_STRING structure pointed to by the *FullPath* parameter, but does not need to initialize this structure. **IoQueryFullDriverPath** assumes that the original contents of this structure are invalid and overwrites them. This routine allocates a string buffer from paged system memory, sets the **Buffer** member of the structure to point to this buffer, and sets the **MaximumLength** and **Buffer** members to describe the buffer and its contents.

The caller is responsible for freeing the storage pointed to by *FullPath*->**Buffer** when the full path string is no longer needed. Typically, the caller frees this storage by calling a routine such as ExFreePool.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DRIVER_OBJECT

ExFreePool

UNICODE_STRING

IoRaiseHardError function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRaiseHardError** routine causes a dialog box to appears that warns the user that a device I/O error has occurred, which might indicate that a physical device is failing.

Syntax

```
void IoRaiseHardError(
  PIRP           Irp,
  PVPB           Vpb,
  PDEVICE_OBJECT RealDeviceObject
);
```

Parameters

`Irp`

Pointer to the IRP that failed because of a device I/O error.

`Vpb`

Pointer to the volume parameter block (VPB), if any, for the mounted file object. This parameter is **NULL** if no VPB is associated with the device object.

`RealDeviceObject`

Pointer to the device object that represents the physical device on which the I/O operation failed.

Return Value

None

Remarks

Highest-level drivers, particularly file system drivers, call **IoRaiseHardError**.

Warning Because **IoRaiseHardError**

```
  uses a normal kernel APC to create a user dialog box, a deadlock can occur if normal kernel APCs are
disabled when a device error occurs. For example:</div>
```

1. An upper-level filter driver calls KeEnterCriticalRegion (which disables normal kernel APCs) and sends an I/O request to a file system driver. The filter driver waits on the completion of the request by the file system driver before the filter driver calls KeLeaveCriticalRegion (which reenables normal kernel APCs).
2. An error occurs on the file system, and the file system driver calls **IoRaiseHardError** to report the error to the user. The file system driver waits on the dialog box.
3. Deadlock now exists: The normal kernel APC created by **IoRaiseHardError** to create the dialog box waits for normal kernel APCs to be enabled. The file system waits on the dialog box before it completes the I/O request. The filter driver waits on completion of the I/O request before it calls **KeLeaveCriticalRegion** (which reenables normal kernel APCs).

The behavior of this routine is dependent of the current state of hard errors for the running thread. If hard errors have been disabled by calling IoSetThreadHardErrorMode, this routine completes the IRP specified by *Irp* without transferring any data into user buffers. In addition, no message is sent to notify the user of this failure.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlIoApcLte, HwStorPortProhibitedDDIs

See Also

IoGetRelatedDeviceObject

IoSetHardErrorOrVerifyDevice

IoSetThreadHardErrorMode

IoRaiseInformationalHardError function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRaiseInformationalHardError** routine sends a dialog box to the user, warning about a device I/O error that indicates why a user I/O request failed.

Syntax

```
BOOLEAN IoRaiseInformationalHardError(
  NTSTATUS         ErrorStatus,
  PUNICODE_STRING String,
  PKTHREAD         Thread
);
```

Parameters

`ErrorStatus`

The error status code (IO_ERR_*XXX*).

`String`

A pointer to a Unicode string, which provides additional information about the error. Some NT status codes require a string parameter, such as a file or directory name. If the specified *ErrorStatus* value does not require a string parameter, set *String* to **NULL**.

`Thread`

A pointer to the thread whose IRP was failed due to the error specified by the *ErrorStatus* parameter.

Return Value

IoRaiseInformationalHardError returns **TRUE** if the dialog box was successfully queued. This routine returns **FALSE** if dialog boxes are disabled for *Thread*, a pool allocation failed, too many dialog boxes are already queued, or an equivalent dialog box is already pending a user response (such as waiting for the user to press RETURN).

Remarks

IoRaiseInformationalHardError takes a system-defined NT error value as a parameter. Driver writers can use the event log APIs to communicate driver-defined event strings to the user.

IoRaiseInformationalHardError behaves as follows:

- If the caller-supplied *ErrorStatus* value is not defined in the Ntstatus.h header file, the dialog box always shows the text "Unknown Hard Error", regardless of the *String* parameter value.
- If the caller-supplied *ErrorStatus* value is defined in the Ntstatus.h header file, the dialog box shows the text in the corresponding MessageText string in Ntstatus.h, regardless of the *String* parameter value.
- If the *Thread* parameter specifies a thread that is running in the context of an application, the text shown in the dialog box is "ApplicationName.exe - System Error".
- If the *Thread* parameter is NULL or if it specifies a thread that is running in an arbitrary system context, the text shown in the dialog box is "System Process - System Error". Additionally, if the *ErrorStatus* value is defined in

Ntstatus.h, the corresponding MessageText string is written to the event log.

If a previous call to the IoSetThreadHardErrorMode routine disabled hard errors for the specified thread, **IoRaiseInformationalHardError** returns **FALSE**.

Starting with Windows Vista, if the routine is called from a thread in session 0 (that is, from any system thread), no dialog box appears when the routine succeeds and returns **TRUE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlIoApcLte, HwStorPortProhibitedDDIs

See Also

IoSetHardErrorOrVerifyDevice

IoSetThreadHardErrorMode

PsGetCurrentThread

IoRegisterBootDriverCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterBootDriverCallback** routine registers a **BOOT_DRIVER_CALLBACK_FUNCTION** routine to be called during the initialization of a boot-start driver and its dependent DLLs.

Syntax

```
PVOID IoRegisterBootDriverCallback(
  PBOOT_DRIVER_CALLBACK_FUNCTION CallbackFunction,
  PVOID                          CallbackContext
);
```

Parameters

`CallbackFunction`

A pointer to the **BOOT_DRIVER_CALLBACK_FUNCTION** routine to be called when initializing a boot-start driver or DLL.

`CallbackContext`

A driver-defined context to be passed to the **BOOT_DRIVER_CALLBACK_FUNCTION** routine pointed to by *CallbackFunction*.

Return Value

A handle that represents the registration. This handle must be supplied as an input parameter in the call to the IoUnRegisterBootDriverCallback routine that unregisters the **BOOT_DRIVER_CALLBACK_FUNCTION** routine.

Remarks

Boot-start drivers must call IoUnRegisterBootDriverCallback and pass the returned handle to unregister the boot-start driver callback before Windows unloads them.

Any error returned from a status update callback is treated as fatal and leads to a system bug check. Additionally, if an initialize image callback returns an error, the driver's image is treated as unknown. To be notified of boot-start driver initialization operations, an *early launch anti-malware* (ELAM) driver can call **IoRegisterBootDriverCallback** to register a **BOOT_DRIVER_CALLBACK_FUNCTION** routine.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

BDCB_CALLBACK_TYPE

BDCB_IMAGE_INFORMATION

IoUnRegisterBootDriverCallback

IoRegisterBootDriverReinitialization function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterBootDriverReinitialization** routine is called by a boot driver to register the driver's reinitialization routine with the I/O manager to be called after all devices have been enumerated and started.

Syntax

```
void IoRegisterBootDriverReinitialization(
  PDRIVER_OBJECT       DriverObject,
  PDRIVER_REINITIALIZE DriverReinitializationRoutine,
  PVOID                Context
);
```

Parameters

`DriverObject`

Pointer to the driver object for the boot driver to be reinitialized.

`DriverReinitializationRoutine`

Pointer to the driver's Reinitialize routine.

`Context`

Optional context pointer to be passed to the driver's reinitialization routine.

Return Value

None

Remarks

A boot driver normally calls **IoRegisterBootDriverReinitialization** from its DriverEntry routine, which is run during boot driver initialization. **IoRegisterBootDriverReinitialization** registers the driver's reinitialization callback routine to be called by the I/O manager after all devices have been enumerated and started. The *DriverReinitializationRoutine* is run in a system thread at IRQL = PASSIVE_LEVEL.

A driver should call **IoRegisterBootDriverReinitialization** only if its *DriverEntry* routine will return STATUS_SUCCESS.

If the *DriverReinitializationRoutine* uses the registry, the *DriverEntry* routine must include in **IoRegisterBootDriverReinitialization**'s *Context* parameter a copy of the string to which *DriverEntry*'s own *RegistryPath* parameter points.

The *DriverEntry* routine can call **IoRegisterBootDriverReinitialization** only once. If the reinitialization routine needs to be run more than once, the *DriverReinitializationRoutine* can call **IoRegisterBootDriverReinitialization** as many additional times as needed, using the *Count* parameter to keep track of the number of times the *DriverReinitializationRoutine* has been called.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntifs.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DRIVER_OBJECT

IoRegisterDriverReinitialization

Reinitialize

IoRegisterDriverReinitialization function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterDriverReinitialization** routine is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.

Syntax

```
void IoRegisterDriverReinitialization(
  PDRIVER_OBJECT       DriverObject,
  PDRIVER_REINITIALIZE DriverReinitializationRoutine,
  PVOID                Context
);
```

Parameters

`DriverObject`

Pointer to the driver object that was input to the DriverEntry routine.

`DriverReinitializationRoutine`

Pointer to the driver's Reinitialize routine.

`Context`

Pointer to the context to be passed to the driver's *Reinitialize* routine.

Return Value

None

Remarks

A driver can call this routine only if its *DriverEntry* routine will return STATUS_SUCCESS. If the driver-supplied Reinitialize routine must use the registry, the *DriverEntry* routine should include a copy of the string to which *RegistryPath* points as part of the context passed to the *Reinitialize* routine in this call.

If the driver is loaded dynamically, it is possible for this to occur during a normally running system, so all references to the reinitialization queue must be synchronized.

The *Count* input to a *DriverReinitializationRoutine* indicates how many times this routine has been called, including the current call.

The *DriverEntry* routine can call **IoRegisterDriverReinitialization** only once. If the *Reinitialize* routine should be run again after any other drivers' *Reinitialize* routines have returned control, the *Reinitialize* routine also can call **IoRegisterDriverReinitialization** as many times as the driver's *Reinitialize* routine should be run.

Usually, a driver with a *Reinitialize* routine is a higher-level driver that controls both PnP and legacy devices. Such a driver must not only create device objects for the devices that the PnP manager detects (and for which the PnP manager calls the driver's AddDevice routine), the driver must also create device objects for legacy devices that the PnP manager does not detect. A driver can use a *Reinitialize* routine to create those device objects and layer the

driver over the next-lower driver for the underlying device.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DRIVER_OBJECT

IoRegisterBootDriverReinitialization

IoReportDetectedDevice function

1/25/2019 • 4 minutes to read • Edit Online

The **IoReportDetectedDevice** routine reports a non-PnP device to the PnP manager.

Syntax

```
NTSTATUS IoReportDetectedDevice(
  PDRIVER_OBJECT                 DriverObject,
  INTERFACE_TYPE                 LegacyBusType,
  ULONG                          BusNumber,
  ULONG                          SlotNumber,
  PCM_RESOURCE_LIST              ResourceList,
  PIO_RESOURCE_REQUIREMENTS_LIST ResourceRequirements,
  BOOLEAN                        ResourceAssigned,
  PDEVICE_OBJECT                 *DeviceObject
);
```

Parameters

`DriverObject`

Pointer to the driver object of the driver that detected the device.

`LegacyBusType`

Specifies the type of bus on which the device resides. The PnP manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

The interface types, such as **PCIBus**, are defined in Wdm.h. If a driver does not know the *LegacyBusType* for the device, the driver supplies the value **InterfaceTypeUndefined** for this parameter.

`BusNumber`

Specifies the bus number for the device. The PnP manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

The bus number distinguishes the bus on which the device resides from other buses of the same type on the computer. The bus-numbering scheme is bus-specific. If a driver does not know the *BusNumber* for the device, the driver supplies the value -1 for this parameter.

`SlotNumber`

Specifies the logical slot number of the device. The PnP manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

If a driver does not know the *SlotNumber* for the device, the driver supplies the value -1 for this parameter.

`ResourceList`

Pointer to the resource list the driver used to detect the device. Resources in this list are in raw, untranslated form.

`ResourceRequirements`

Optionally points to a resource requirements list for the detected device. **NULL** if the caller does not have this information for the device.

`ResourceAssigned`

Specifies whether the device's resources have already been reported to the PnP manager. If*ResourceAssigned* is **TRUE**, the resources have already been reported, possibly with IoReportResourceForDetection, and the PnP manager will not attempt to claim them on behalf of the device. If **TRUE**, the PnP manager will also not claim resources when the device is root-enumerated on subsequent boots.

`DeviceObject`

Optionally points to a PDO for the detected device.

NULL if the caller does not have a PDO for the device, which is typically the case. If *DeviceObject* is **NULL**, the PnP manager creates a PDO for the device and returns a pointer to the caller.

If the caller supplies a PDO, the PnP manager does not create a new PDO. On a given call to this routine the *DeviceObject* parameter is either an IN or an OUT parameter, but not both.

Return Value

IoReportDetectedDevice returns STATUS_SUCCESS on success, or the appropriate error code on failure.

Remarks

Drivers for legacy devices use **IoReportDetectedDevice** to report their devices to the system. A driver should only call **IoReportDetectedDevice** to report a legacy, non-PnP device. PnP devices should be reported in response to an IRP_MN_QUERY_DEVICE_RELATIONS request.

Drivers only need to call **IoReportDetectedDevice** the first time they are loaded because the PnP manager caches the reported information. Drivers that use this routine should store a flag in the registry to indicate whether they have already done device detection.

A driver typically calls this routine from its DriverEntry routine. A few drivers, like certain NDIS or EISA drivers, might call this routine from an AddDevice routine.

On successful completion of **IoReportDetectedDevice**, the caller should attach an FDO to the PDO returned at *DeviceObject*. Once the caller attaches its FDO, the caller is the function driver for the device, at least temporarily. There are no filter drivers. The PnP manager owns the PDO.

The PnP manager considers the device to be started and therefore does not call the driver's AddDevice routine and does not send an IRP_MN_START_DEVICE request. The driver must be prepared to handle all other PnP IRPs, however.

IoReportDetectedDevice marks the device as a root-enumerated device and this identification is persistent across system boots. During subsequent system boots the PnP manager "detects" the device on the root-enumerated list and configures it like a PnP device: the PnP manager queries for device information, identifies the appropriate drivers and calls their *AddDevice* routines, and sends all the appropriate PnP IRPs.

The system generates two compatible ID strings for the device, of the form DETECTED*Interface*<i>Driver and DETECTED<i>Driver. *Interface* is the string name of the INTERFACE_TYPE of the first bus specified in the *ResourceList* parameter. *Interface* is set to "Internal" if no bus is specified. *Driver* is the driver's service name. A driver can provide additional hardware IDs or compatible IDs by handling the IRP_MN_QUERY_ID request.

A driver writer must provide an INF file that matches any of the specified hardware IDs or compatible IDs. The INF file should specify the original driver that called **IoReportDetectedDevice** as the driver to load for those IDs. The system uses this information to rebuild the driver stack for the device, for example on restart. Callers of **IoReportDetectedDevice** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IRP_MN_QUERY_DEVICE_RELATIONS

IoReportResourceForDetection

IoReportResourceForDetection function

1/25/2019 • 3 minutes to read • Edit Online

The **IoReportResourceForDetection** routine claims hardware resources in the configuration registry for a legacy device.

Syntax

```
NTSTATUS IoReportResourceForDetection(
  PDRIVER_OBJECT    DriverObject,
  PCM_RESOURCE_LIST DriverList,
  ULONG             DriverListSize,
  PDEVICE_OBJECT    DeviceObject,
  PCM_RESOURCE_LIST DeviceList,
  ULONG             DeviceListSize,
  PBOOLEAN          ConflictDetected
);
```

Parameters

`DriverObject`

Pointer to the driver object that was input to the driver's DriverEntry routine.

`DriverList`

Optionally points to a caller-supplied buffer that contains the driver's resource list, if the driver claims the same resources for all its devices. If the caller specifies a *DeviceList*, this parameter is ignored.

`DriverListSize`

Specifies the size in bytes of an optional *DriverList*. If *DriverList* is **NULL**, this parameter should be zero.

`DeviceObject`

Optionally points to the device object representing device for which the driver is attempting to claim resources.

`DeviceList`

Optionally points to a caller-supplied buffer containing the device's resource list. If the driver claims the same resources for all its devices, the caller can specify a *DriverList* instead.

`DeviceListSize`

Specifies the size in bytes of an optional *DeviceList*. If *DeviceList* is **NULL**, this parameter should be zero.

`ConflictDetected`

Pointer to a caller-supplied Boolean value that is set to **TRUE** on return if the resources are not available.

Return Value

IoReportResourceForDetection returns STATUS_SUCCESS if the resources are claimed. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_CONFLICTING_ADDRESSES	The resources could not be claimed because they are already in use or are needed for a PnP-enumerable device.
STATUS_UNSUCCESSFUL	The *DeviceList* or *DriverList* is invalid.

Remarks

This routine is for drivers that detect earlier hardware which cannot be enumerated by Plug and Play (PnP).

If a driver supports only PnP hardware, it does no detection and therefore does not call **IoReportResourceForDetection**. The PnP system enumerates each PnP device, assigns resources to the device, and passes those resources to the device's drivers in an IRP_MN_START_DEVICE request.

If a PnP driver supports legacy hardware, however, it must call **IoReportResourceForDetection** to claim hardware resources before it attempts to detect the device.

Callers of this routine specify a CM_RESOURCE_LIST in either a *DeviceList* or a *DriverList* that is allocated from paged memory and that consists of raw, untranslated resources. The caller is responsible for freeing the memory.

A driver that can control more than one legacy card at the same time should claim the resources for each device against the device object for the respective device (using the *DeviceObject*, *DeviceList*, and *DeviceListSize* parameters). Such a driver must not claim these resources against their driver object.

A **CM_RESOURCE_LIST** contains two variable-sized arrays. Each array has a default size of one. If either array has more than one element, the caller must allocate memory dynamically to contain the additional elements. Only one CM_PARTIAL_RESOURCE_DESCRIPTOR can be part of each CM_FULL_RESOURCE_DESCRIPTOR in the list, except for the last full resource descriptor in the **CM_RESOURCE_LIST**, which can have additional partial resource descriptors in its array.

IoReportResourceForDetection, with the help of the PnP manager, determines whether the resources being requested conflict with resources that have already been claimed.

If a conflict is detected, this routine sets the BOOLEAN at *ConflictDetected* to **TRUE** and returns STATUS_CONFLICTING_ADDRESSES.

If no conflict is detected, this routine claims the resources, sets the BOOLEAN at *ConflictDetected* to **FALSE**, and returns STATUS_SUCCESS.

If this routine succeeds and the driver detects a legacy device, the driver reports the device to the PnP manager by calling IoReportDetectedDevice. In that call, the driver sets *ResourceAssigned* to **TRUE** so the PnP manager does

not attempt to claim the resources again.

When a driver no longer requires the resources claimed by a call to this routine, the driver calls this routine again with a *DriverList* or *DeviceList* with a **Count** of zero.

If a driver claims resources on a device-specific basis for more than one device, the driver must call this routine for each such device.

A driver can call this routine more than once for a given device. If one set of resources fails, the driver can call the routine again for the same device with a different set of resources. If a set of resources succeeds, the driver can call this routine again with a new list; the new list replaces the previous list.

Callers of **IoReportResourceForDetection** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)

See Also

CM_RESOURCE_LIST

IoReportDetectedDevice

IoReportRootDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReportRootDevice** routine reports a device that cannot be detected by a PnP bus driver to the PnP Manager. **IoReportRootDevice** allows only one device per driver to be created.

Syntax

```
NTSTATUS IoReportRootDevice(
  PDRIVER_OBJECT DriverObject
);
```

Parameters

`DriverObject`

Pointer to the driver object of the driver that detected the device.

Return Value

IoReportRootDevice returns STATUS_SUCCESS on success, or the appropriate error code on failure.

Remarks

Drivers for devices that cannot be detected by a PnP bus driver use **IoReportRootDevice** to report their device to the system. Devices that can be detected by a PnP bus driver should be reported in response to an IRP_MN_QUERY_DEVICE_RELATIONS request.

A driver typically calls this routine from its DriverEntry routine.

IoReportRootDevice marks the device as a root-enumerated device and this identification is persistent across system boots. The PnP manager "detects" the device on the root-enumerated list and configures it like a PnP device: the PnP manager queries for device information, identifies the appropriate drivers and calls their *AddDevice* routines, and sends all the appropriate PnP IRPs.

The system generates a single Hardware ID string for the device, of the form ROOT<i>Driver where *Driver* is the driver's service name. A driver can provide additional hardware IDs or compatible IDs by handling the IRP_MN_QUERY_ID request.

A driver writer must provide an INF file that matches any of the specified hardware IDs or compatible IDs. The INF file should specify the original driver that called **IoReportRootDevice** as the driver to load for those IDs. The system uses this information to rebuild the driver stack for the device, for example on restart. Callers of **IoReportRootDevice** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

> **Note** Drivers should use IoReportDetectedDevice to report hardware devices that use resources but cannot be detected by a PnP bus driver instead of **IoReportRootDevice**.

Requirements

Minimum supported client	Available in Windows 8 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)

See Also

AddDevice

DriverEntry

IRP_MN_QUERY_DEVICE_RELATIONS

IRP_MN_QUERY_ID

IRP_MN_START_DEVICE

IoReportDetectedDevice

IoReportResourceForDetection

IoSetActivityIdIrp function

1/25/2019 • 2 minutes to read • Edit Online

The IoSetActivityIdIrp routine associates an activity ID with an IRP.

Syntax

```
NTSTATUS IoSetActivityIdIrp(
  PIRP    Irp,
  LPCGUID Guid
);
```

Parameters

`Irp`

The IRP to associate the activity ID with.

`Guid`

A pointer to the GUID that represents the ID to store in the IRP. If NULL, IoSetActivityIdIrp attempts to retrieve the activity ID from the current thread if it was the thread that originally issued the request.

Return Value

IoSetActivityIdIrp returns STATUS_SUCCESS if the call is successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_NOT_SUPPORTED	No GUID was provided and the ETW activity ID was unavailable.
STATUS_UNSUCCESSFUL	The I/O tracing provider has not been enabled on the IRP.

Remarks

Drivers should use IoSetActivityIdIrp only on IRPs that have been allocated using IoAllocateIrp (and freed using IoFreeIrp). Otherwise, memory leakage may result.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level if a GUID is passed in, otherwise PASSIVE_LEVEL.

IoSetActivityIdThread function

1/25/2019 • 2 minutes to read • Edit Online

The IoSetActivityIdThread routine associates an activity ID with the current thread. Drivers should use this routine when they are tracing aware and are issuing I/O on a worker thread.

Syntax

```
LPCGUID IoSetActivityIdThread(
  LPCGUID ActivityId
);
```

Parameters

`ActivityId`

The activity ID provided by caller.

Return Value

The activity ID that was previously set on the thread. Drivers must call IoClearActivityIdThread with this pointer when tracing is completed within the same thread context.

Remarks

Drivers that use I/O work items do not need to call this routine because the I/O subsystem takes care of propagating activity IDs to threads in that case.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

IoSetHardErrorOrVerifyDevice function

1/25/2019 • 2 minutes to read • Edit Online

Lower-level drivers call the **IoSetHardErrorOrVerifyDevice** routine to identify a removable media device that has encountered an error, so that a file system driver can prompt the user to verify that the medium is valid.

Syntax

```
void IoSetHardErrorOrVerifyDevice(
  PIRP           Irp,
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`Irp`

Pointer to the IRP for which the driver encountered a user-induced error.

`DeviceObject`

Pointer to the target device to be verified for the I/O operation.

Return Value

None

Remarks

Lower-level drivers for removal media devices must call **IoSetHardErrorOrVerifyDevice** before completing an IRP, if a call to IoIsErrorUserInduced returns **TRUE**. For more information, see Supporting Removable Media.

Subsequently, a file system driver will use information supplied in the specified device object to send a dialog box to the user, who can correct the error and retry the operation or cancel it.

The IRP passed to **IoSetHardErrorOrVerifyDevice** must be associated with a thread. Therefore, before a driver calls **IoSetHardErrorOrVerifyDevice**, it must verify that *irp*->**Tail.Overlay**.Thread is not **NULL**. If the value is **NULL**, **IoSetHardErrorOrVerifyDevice** must not be called using the current IRP. Instead, the driver should check all IRPs subsequently received and processed, until one is received for which *irp*->**Tail.Overlay**.Thread is not **NULL**. At that point, **IoSetHardErrorOrVerifyDevice** can be called.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IoIsErrorUserInduced

IoRaiseHardError

IoRaiseInformationalHardError

IoSetMasterIrpStatus function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetMasterIrpStatus** routine conditionally replaces the **Status** value in an IRP with the specified NTSTATUS value.

Syntax

```
void IoSetMasterIrpStatus(
  PIRP     MasterIrp,
  NTSTATUS Status
);
```

Parameters

`MasterIrp`

A pointer to the master IRP. For more information, see Remarks.

`Status`

An NTSTATUS value to compare to the **Status** member of the I/O status block in the master IRP.

Return Value

None.

Remarks

On receipt of an IRP, a driver can create two or more subordinate IRPs to perform the work requested by the original (or master) IRP. When the subordinate IRPs complete, the driver gathers the completion status codes from the subordinate IRPs and merges them to form a single completion status code for the master IRP.

IoSetMasterIrpStatus implements a uniform policy for merging the status codes from multiple subordinate IRPs into a status code for a master IRP. For example, if a master IRP is split into two subordinate IRPs, and one of these subordinate IRPs succeeds and the other fails, the status code from the failing IRP is used as the status code for the master IRP. However, if both subordinate IRPs fail, the more severe failure code is used as the status for the master IRP.

Before the first call to **IoSetMasterIrpStatus**, the driver sets the **IoStatus.Status** member in the master IRP to STATUS_SUCCESS (or to STATUS_FT_READ_FROM_COPY in case it is expected). Next, as each subordinate IRP completes, the driver calls **IoSetMasterIrpStatus** to merge the status code from this IRP with the status code in the master IRP. In this call, the *PIRP* parameter points to the master IRP, and the *Status* parameter is set to the value of the **IoStatus.Status** member in the subordinate IRP.

As a general rule, **IoSetMasterIrpStatus** replaces the status code in *PIRP*->**IoStatus.Status** with the *Status* value in the following cases:

- The status value in **PIRP* is STATUS_SUCCESS, and *Status* is an error code.
- The status value in **PIRP* is an error code, but *Status* is a more severe error code.

There are two exceptions to the general rule. A *Status* value of STATUS_VERIFY_REQUIRED always replaces the status

value in **PIRP*. A *Status* value of STATUS_FT_READ_FROM_COPY never replaces the status value in **PIRP*.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level.

See Also

IRP

IoSetSystemPartition function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetSystemPartition** routine sets the boot partition for the system.

Syntax

```
NTSTATUS IoSetSystemPartition(
  PUNICODE_STRING VolumeNameString
);
```

Parameters

`VolumeNameString`

Pointer to a Unicode string that specifies the MS-DOS name of the system partition.

Return Value

STATUS_SUCCESS if the boot partition can be set, or an error code on failure.

Remarks

The specified partition must contain the boot loader.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

IoSetThreadHardErrorMode function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetThreadHardErrorMode** routine enables or disables hard error reporting for the current thread.

Syntax

```
BOOLEAN IoSetThreadHardErrorMode(
  BOOLEAN EnableHardErrors
);
```

Parameters

`EnableHardErrors`

Specifies whether hard error reporting to the user should be enabled or disabled for this thread. A value of **TRUE** enables hard error reporting. **FALSE** disables it.

Return Value

IoSetThreadHardErrorMode returns **TRUE** if hard errors were enabled from this thread before this routine completed execution. Otherwise, this routine returns **FALSE**.

Remarks

If hard errors are disabled for a given thread, calls to **IoRaiseHardError** will not display a message to the user indicating that a serious error has occurred. In addition, the IRP that is passed to **IoRaiseHardError** is completed without any data being copied into user buffers. Calling **IoRaiseInformationalHardError** after disabling hard errors causes that routine to always return **FALSE** for this thread.

Requirements

Minimum supported client	Available in Microsoft Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IoRaiseHardError

IoRaiseInformationalHardError

IoTransferActivityId function

1/25/2019 • 2 minutes to read • Edit Online

The IoTransferActivityId routine logs an ETW transfer event using the I/O tracing provider on behalf of the caller. This allows a driver to associate two related activity IDs without requiring a specific provider to be enabled.

Syntax

```
void IoTransferActivityId(
  LPCGUID ActivityId,
  LPCGUID RelatedActivityId
);
```

Parameters

`ActivityId`

The source activity ID.

`RelatedActivityId`

The new activity ID to be transferred from the source activity ID.

Return Value

None.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

IoUnregisterBootDriverCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUnRegisterBootDriverCallback** routine unregisters a previously registered **BOOT_DRIVER_CALLBACK_FUNCTION** routine.

Syntax

```
void IoUnregisterBootDriverCallback(
  PVOID CallbackHandle
);
```

Parameters

`CallbackHandle`

A handle returned from a previous call to IoRegisterBootDriverCallback.

Return Value

None

Remarks

A driver that calls IoRegisterBootDriverCallback must call **IoUnRegisterBootDriverCallback** before being unloaded.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	<= APC_LEVEL

See Also

IoRegisterBootDriverCallback

IoVerifyPartitionTable function

1/25/2019 • 2 minutes to read • Edit Online

The **IoVerifyPartitionTable** routine checks the validity of the partition table for a disk.

Syntax

```
NTSTATUS IoVerifyPartitionTable(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        FixErrors
);
```

Parameters

`DeviceObject`

Specifies the device object of the disk to verify.

`FixErrors`

Specifies if the routine should correct any correctable errors in the partition table.

Return Value

The routine returns STATUS_SUCCESS if the partition table is error-free.

If the caller specifies **FALSE** for the *FixErrors* parameter and the partition table has errors, the routine returns STATUS_DISK_CORRUPT_ERROR. If the caller specifies **TRUE** for *FixErrors*, and the routine is able to correct all errors in the partition table, it returns STATUS_SUCCESS. Otherwise it returns STATUS_DISK_CORRUPT_ERROR.

The routine might also return other NT status error codes.

Remarks

IoVerifyPartitionTable must be used only by disk drivers.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= APC_LEVEL"

IoVolumeDeviceToDosName function

1/25/2019 • 2 minutes to read • Edit Online

The **IoVolumeDeviceToDosName** routine returns the MS-DOS path for a specified device object that represents a file system volume.

Syntax

```
NTSTATUS IoVolumeDeviceToDosName(
  PVOID           VolumeDeviceObject,
  PUNICODE_STRING DosName
);
```

Parameters

`VolumeDeviceObject`

A pointer to a device object that represents a volume device object created by a storage class driver.

`DosName`

A pointer to a caller-allocated UNICODE_STRING structure. If the call is successful, **IoVolumeDeviceToDosName** sets the values of the **Length**, **MaximumLength**, and **Buffer** members of this structure. On exit, the **Buffer** member points to a wide-character, null-terminated string that contains the MS-DOS path of the volume device object specified by *VolumeDeviceObject*. For more information, see Remarks.

Return Value

IoVolumeDeviceToDosName returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The routine failed due to invalid parameter values passed by the caller.

STA TUS _INS UFFI CIE NT_ RES OU RCE S	The routine failed to allocate resources required for this operation.

Remarks

IoVolumeDeviceToDosName allocates the string buffer pointed to by the **Buffer** member of the UNICODE_STRING structure that the *DosName* parameter points to. After this buffer is no longer required, a caller of this routine should call the ExFreePool routine to free the buffer.

Starting with Windows Vista, you must ensure that APCs are not disabled before calling this routine. The KeAreAllApcsDisabled routine can be used to verify that APCs are not disabled.

Requirements

Minimum supported client	Available starting with Windows XP. Drivers that must work in earlier versions of Windows NT-based operating systems can use RtlVolumeDeviceToDosName, which behaves identically.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ExFreePool

KeAreAllApcsDisabled

UNICODE_STRING

KeAreApcsDisabled function

1/25/2019 • 2 minutes to read • Edit Online

The **KeAreApcsDisabled** routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.

Syntax

```
BOOLEAN KeAreApcsDisabled(

);
```

Parameters

This function has no parameters.

Return Value

KeAreApcsDisabled returns **TRUE** if the thread is within a critical region or a guarded region, and **FALSE** otherwise.

Remarks

A thread running at IRQL = PASSIVE_LEVEL can use **KeAreApcsDisabled** to determine if normal kernel APCs are disabled. A thread that is inside a critical region has both user APCs and normal kernel APCs disabled, but not special kernel APCs. A thread that is inside a guarded region has all APCs disabled, including special kernel APCs.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAreAllApcsDisabled

KeEnterCriticalRegion

KeLeaveCriticalRegion

KeBugCheck function

1/25/2019 • 2 minutes to read • Edit Online

The **KeBugCheck** routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.

Syntax

```
void KeBugCheck(
  ULONG BugCheckCode
);
```

Parameters

`BugCheckCode`

Specifies a value that indicates the reason for the bug check.

Return Value

None

Remarks

A bug check is a system-detected error that causes an immediate, controlled shutdown of the system. Various kernel-mode components perform run-time consistency checking. When such a component discovers an unrecoverable inconsistency, it causes a bug check to be generated.

Whenever possible, all kernel-mode components should log an error and continue to run, rather than calling **KeBugCheck**. For example, if a driver is unable to allocate required resources, it should log an error so that the system continues to run; it must not generate a bug check.

A driver or other kernel-mode component should call this routine only in cases of a fatal, unrecoverable error that could corrupt the system itself.

KeBugCheck can be useful in the early stages of developing a driver, or while it is undergoing testing. In these circumstances, the *BugCheckCode* passed to this routine should be distinct from those codes already in use by Windows or its drivers. For a list of these codes, see Bug Check Codes.

However, even during driver development, this routine is of only limited utility, since it results in a complete system shutdown. A more effective debugging method is to attach a kernel debugger to the system and then use routines that send messages to the debugger or break into the debugger. For further information, see Using Debugging Code in a Driver.

When a bug check is unavoidable, most system components call KeBugCheckEx, which provides more information about the cause of such an inconsistency than **KeBugCheck**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoAllocateErrorLogEntry

IoWriteErrorLogEntry

KeBugCheckEx

KeRegisterBugCheckCallback

KeRegisterBugCheckReasonCallback

KeEnterCriticalRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeEnterCriticalRegion** routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.

Syntax

```
void KeEnterCriticalRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

A driver calls this routine to enter a critical region in which the execution of normal kernel APCs is deferred until this driver exits the critical region by calling the KeLeaveCriticalRegion routine. Any caller of **KeEnterCriticalRegion** should call **KeLeaveCriticalRegion** as quickly as possible after entering a critical region.

Highest-level drivers can call **KeEnterCriticalRegion** while running in the context of the thread that requested the current I/O operation.

A thread that is inside a critical region has both user APCs and normal kernel APCs disabled, but not special kernel APCs. For more information about these APC types, see Types of APCs.

Critical regions can be entered recursively and each call to **KeEnterCriticalRegion** must have a matching call to **KeLeaveCriticalRegion**.

A driver can use a critical region to acquire and release exclusive access to a shared resource. In this case, the ExEnterCriticalRegionAndAcquireResourceExclusive and ExReleaseResourceAndLeaveCriticalRegion routines can be used instead of the **KeEnterCriticalRegion** and **KeLeaveCriticalRegion** routines. For more information, see the code example in ExEnterCriticalRegionAndAcquireResourceExclusive.

For more information about APCs, see Asynchronous Procedure Calls.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	CriticalRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExEnterCriticalRegionAndAcquireResourceExclusive

ExReleaseResourceAndLeaveCriticalRegion

KeAreApcsDisabled

KeLeaveCriticalRegion

KeEnterGuardedRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeEnterGuardedRegion** routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.

Syntax

```
void KeEnterGuardedRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

To exit a guarded region entered with **KeEnterGuardedRegion**, call the **KeLeaveGuardedRegion** routine. Guarded regions can be nested. APCs are not reenabled until the thread exits the outermost guarded region.

For more information about guarded regions, see Critical Regions and Guarded Regions.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	GuardedRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs

See Also

KeLeaveGuardedRegion

KeExpandKernelStackAndCallout function

1/25/2019 • 2 minutes to read • Edit Online

The **KeExpandKernelStackAndCallout** routine calls a routine with a guaranteed amount of stack space.

Syntax

```
NTSTATUS KeExpandKernelStackAndCallout(
  PEXPAND_STACK_CALLOUT Callout,
  PVOID                 Size,
  SIZE_T                Parameter
);
```

Parameters

`Callout`

Pointer to an ExpandedStackCall routine.

`Size`

Specifies the number of bytes on the stack to provide to the *ExpandedStackCall* routine. This value must be large enough to accommodate the stack usage of the *ExpandedStackCall* routine and any call that this routine might make. This value must not exceed MAXIMUM_EXPANSION_SIZE.

`Parameter`

Specifies the parameter to pass to the *ExpandedStackCall* routine.

Return Value

KeExpandKernelStackAndCallout returns STATUS_SUCCESS if the operation succeeds or an appropriate NTSTATUS value if the operation fails.

Remarks

KeExpandKernelStackAndCallout expands the kernel stack by *Size* bytes for use by the ExpandedStackCall routine. If there is not enough space available on the stack, **KeExpandKernelStackAndCallout** allocates a new kernel stack segment. The routine then calls the *ExpandedStackCall* routine.

In Windows 7, Windows Server 2008 R2, and later versions of Windows, consider using the KeExpandKernelStackAndCalloutEx routine instead of **KeExpandKernelStackAndCallout**. **KeExpandKernelStackAndCalloutEx** is similar to **KeExpandKernelStackAndCallout** but has additional parameters and can be called at IRQL <= DISPATCH_LEVEL.

The calling thread must not call the PsTerminateSystemThread routine until the thread's *ExpandedStackCall* routine returns. **PsTerminateSystemThread** checks to determine if the *ExpandedStackCall* routine is still active and, if it is, causes a bug check.

Requirements

Minimum supported client	Available starting with Windows Server 2003 on x64-based processors, and starting with Windows Vista on all processors.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

ExpandedStackCall

KeExpandKernelStackAndCalloutEx

PsTerminateSystemThread

KeGetCurrentNodeNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentNodeNumber** routine gets the NUMA node number for the logical processor that the caller is running on.

Syntax

```
USHORT KeGetCurrentNodeNumber(

);
```

Parameters

This function has no parameters.

Return Value

KeGetCurrentNodeNumber returns the node number.

Remarks

In a non-uniform memory access (NUMA) multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

In a NUMA multiprocessor system that contains *n* nodes, the nodes are numbered from 0 to *n*-1. To get the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryHighestNodeNumber

KeGetCurrentProcessorNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentProcessorNumber** routine returns the system-assigned number of the current processor on which the caller is running.

Syntax

```
ULONG KeGetCurrentProcessorNumber(

);
```

Parameters

This function has no parameters.

Return Value

KeGetCurrentProcessorNumber returns a ULONG value that represents the processor on which the caller is currently running.

Remarks

KeGetCurrentProcessorNumber can be called to debug spin lock usage on SMP machines during driver development. A driver also might call **KeGetCurrentProcessorNumber** if it maintained some per-processor data and attempted to reduce cache-line contention.

The number of processors in an SMP machine is a zero-based value.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeGetCurrentProcessorNumberEx routine, which specifies a processor group, instead of **KeGetCurrentProcessorNumber**, which does not. However, the implementation of **KeGetCurrentProcessorNumber** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows. In this implementation, **KeGetCurrentProcessorNumber** returns the group-relative processor number if the caller is running on a processor in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support processor groups. If the caller is running on a processor in any group other than group 0, this routine returns a number that is less than the number of processors in group 0. This behavior ensures that the return value is less than the return value of the KeQueryActiveProcessorCount routine.

If the call to **KeGetCurrentProcessorNumber** occurs at IRQL <= APC_LEVEL, a processor switch can occur between instructions. Consequently, callers of **KeGetCurrentProcessorNumber** usually run at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Desktop
Header	ntddk.h (include Ntddk.h)
IRQL	See Remarks section.
DDI compliance rules	Irql_Miscellaneous_Function

See Also

KeGetCurrentProcessorNumberEx

KeQueryActiveProcessorCount

KeGetCurrentProcessorNumberEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentProcessorNumberEx** routine gets the processor number of the logical processor that the caller is running on.

Syntax

```
ULONG KeGetCurrentProcessorNumberEx(
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcNumber`

A pointer to a caller-allocated buffer into which the routine writes a PROCESSOR_NUMBER structure that identifies the current logical processor. This structure contains the group number and the group-relative processor number. You can set this parameter to **NULL** if you do not need this information.

Return Value

KeGetCurrentProcessorNumberEx returns the systemwide processor index of the logical processor that the caller is running on.

Remarks

If *ProcNumber* is non-**NULL**, the buffer that is pointed to by *ProcNumber* must be large enough to contain a PROCESSOR_NUMBER structure. In contrast to this structure, which contains a group number and a group-relative processor number, the return value is a processor index that identifies the processor across the entire multiprocessor system.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

A related routine, KeGetCurrentProcessorNumber, returns the current processor number, but this routine, unlike **KeGetCurrentProcessorNumberEx**, does not provide a group number for the processor. In Windows 7 and later versions of the Windows operating system, **KeGetCurrentProcessorNumber** returns the group-relative processor number if the caller is running on a processor in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. However, if the caller is running on a processor in any group other than group 0, this routine returns a number that is less than the number of processors in group 0. This behavior ensures that the return value is less than the return value of the KeQueryActiveProcessorCount routine. Existing drivers that call **KeGetCurrentProcessorNumber** and that use no group-oriented features run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeGetCurrentProcessorNumberEx** instead of **KeGetCurrentProcessorNumber**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentProcessorNumber

KeQueryActiveProcessorCount

PROCESSOR_NUMBER

KeInitializeCrashDumpHeader function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeCrashDumpHeader** routine supplies the header information the system requires for a crash dump file.

Syntax

```
NTSTATUS KeInitializeCrashDumpHeader(
  ULONG  DumpType,
  ULONG  Flags,
  PVOID  Buffer,
  ULONG  BufferSize,
  PULONG BufferNeeded
);
```

Parameters

`DumpType`

Specifies the type of dump file. The only valid value is DUMP_TYPE_FULL.

`Flags`

Specifies flags for the dump file. The only valid value is 0.

`Buffer`

Pointer to the buffer that receives the header information.

`BufferSize`

Specifies the size in bytes of the buffer pointed to by *Buffer*.

`BufferNeeded`

Optionally, a pointer to a variable that receives the size necessary to hold the complete header information.

Return Value

KeInitializeCrashDumpHeader returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

Drivers can use this routine to manually create a crash dump file. The file can be created at any time, and used by a debugger to examine the state of the system.

To create a crash-dump file, call **KeInitializeCrashDumpHeader** to create the header, then append the contents of memory to the header. Note that the driver is not required to record the contents of memory immediately after calling the routine: the header can normally be created at any time before the crash-dump file is written.

Because the routine is designed to be called well before the memory contents are recorded, it has the following limitations:

- The routine does not record any information about active exception records.
- If the size of system RAM changes, the header must be recreated.

The routine does not record any secondary dump data.

Starting with Windows 8, **KeInitializeCrashDumpHeader** always writes the base address of the system process page directory to the crash-dump header, regardless of the process context from which **KeInitializeCrashDumpHeader** is called. The debugger can use this directory to access the crash-dump file in the context of the system process.

In earlier versions of Windows, **KeInitializeCrashDumpHeader** writes the base address of the page directory of the caller's current process context to the crash-dump header. Thus, **KeInitializeCrashDumpHeader** must be called from the system process. Otherwise, the debugger will be unable to access the crash-dump file in the process context in which the file was saved.

Starting with Windows 8, **KeInitializeCrashDumpHeader** is declared in the Wdm.h header file in the Windows Driver Kit (WDK). To use this routine with earlier versions of the WDK, include the following function declaration in your driver code:

C++

```
NTSTATUS
  KeInitializeCrashDumpHeader(
    __in ULONG  DumpType,
    __in ULONG  Flags,
    __out PVOID  Buffer,
    __in ULONG  BufferSize,
    __out_opt PULONG  BufferNeeded
    );
```

Requirements

Minimum supported client	Available starting with Windows Server 2003 with SP1.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

KeInvalidateAllCaches function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInvalidateAllCaches** routine flushes all processor caches.

Syntax

```
BOOLEAN KeInvalidateAllCaches(

);
```

Parameters

This function has no parameters.

Return Value

KeInvalidateAllCaches returns a value that is reserved for system use. Treat this value as VOID.

Remarks

The **KeInvalidateAllCaches** routine flushes each processor's caches and marks each cache's contents invalid. The processor caches are guaranteed to have completed the flush operation before **KeInvalidateAllCaches** returns.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

KeLeaveCriticalRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeLeaveCriticalRegion** routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to **KeEnterCriticalRegion**.

Syntax

```
void KeLeaveCriticalRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

Highest-level drivers can call this routine while running in the context of the thread that requested the current I/O operation.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	CriticalRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

KeAreApcsDisabled

KeEnterCriticalRegion

KeLeaveGuardedRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeLeaveGuardedRegion** routine exits a guarded region entered by **KeEnterGuardedRegion**.

Syntax

```
void KeLeaveGuardedRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

Guarded regions can be nested, so APCs are only reenabled once the thread exists the outermost guarded region.

For more information about guarded regions, see Critical Regions and Guarded Regions.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	GuardedRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs

See Also

KeEnterGuardedRegion

KePulseEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KePulseEvent** routine atomically sets an event object to a signaled state, attempts to satisfy as many waits as possible, and then resets the event object to a not-signaled state.

Syntax

```
LONG KePulseEvent(
  PRKEVENT  Event,
  KPRIORITY Increment,
  BOOLEAN   Wait
);
```

Parameters

`Event`

A pointer to a dispatcher object of type KEVENT.

`Increment`

Specifies the priority increment that is to be applied if setting the event causes a wait to be satisfied.

`Wait`

Specifies a Boolean value that signifies whether the call to **KePulseEvent** will be immediately followed by a call to one of the **KeWait*Xxx*** routines. If **TRUE**, the **KePulseEvent** call is immediately followed by a call to KeWaitForMultipleObjects, KeWaitForMutexObject, or KeWaitForSingleObject. For more information, see the following Remarks section.

Return Value

The previous signal state of the event object.

Remarks

For more information about event objects, see Event Objects.

The **KePulseEvent** routine might temporarily raise the IRQL. If the *Wait* parameter is **FALSE**, the routine, before it returns, restores the IRQL to the original value that it had at the start of the call.

If *Wait* = **TRUE**, the routine returns without lowering the IRQL. In this case, the **KePulseEvent** call must be immediately followed by a **KeWait*Xxx*** call. By setting *Wait* = **TRUE**, the caller can prevent an unnecessary context switch from occurring between the **KePulseEvent** call and the **KeWait*Xxx*** call. The **KeWait*Xxx*** routine, before it returns, restores the IRQL to its original value at the start of the **KePulseEvent** call. Although the IRQL disables context switches between the two calls, these calls cannot reliably be used as the start and end of an atomic operation. For example, between these two calls, a thread that is running at the same time on another processor might change the state of the event object or of the target of the wait.

If the caller is executing at IRQL = DISPATCH_LEVEL or in an arbitrary thread context, the *Timeout* parameter to **KeWait*Xxx*** must be zero.

Warning If a thread waiting for *Event* is currently running a kernel APC, then, when **KePulseEvent** is called, this thread's wait is not satisfied. After the kernel APC completes, the thread remains in the wait state.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

KeReadStateEvent

KeResetEvent

KeSetEvent

KeWaitForMultipleObjects

KeWaitForMutexObject

KeWaitForSingleObject

KeQueryActiveGroupCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveGroupCount** routine returns the number of active processor groups in a multiprocessor system.

Syntax

```
USHORT KeQueryActiveGroupCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryActiveGroupCount returns a count of the active groups.

Remarks

A processor group is active if it contains one or more active logical processors.

An active logical processor is a logical processor that Windows has started up and added to a multiprocessor system. The term *active processor* applies to a processor that is available to perform processing work, regardless of whether the processor is currently performing processing work or is idle. In some systems, the number of active processors might remain unchanged from system startup to shutdown. In other systems, Windows might dynamically add active processors while the system is running. Windows never removes active processors from a system.

Thus, the number of active processors in a multiprocessor system can increase between system startup and shutdown, but this number never decreases. Similarly, the number of active groups in the system can increase if a processor is dynamically added to an inactive group while the system is running. The number of active groups never decreases.

In contrast, the value that is returned by KeQueryMaximumGroupCount remains constant during runtime. This value is the maximum number of groups that the system can have.

Call the KeQueryActiveProcessorCountEx routine to determine the number of active logical processors in a particular group.

Windows 7 is the first version of Windows to support processor groups. In Windows 7, only 64-bit versions of Windows support multiple groups; 32-bit versions of Windows support only one group. If a multiprocessor system is running a 64-bit version of Windows and contains no more than 64 logical processors, Windows assigns all processors to group 0. A multiprocessor system that is running a 32-bit version of Windows can contain no more than 32 processors.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryMaximumGroupCount

KeQueryActiveProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessorCount** routine returns the number of currently active processors.

Syntax

```
ULONG KeQueryActiveProcessorCount(
  PKAFFINITY ActiveProcessors
);
```

Parameters

`ActiveProcessors`

A pointer to a **KAFFINITY**-typed variable into which the routine writes a bitmask that represents the set of currently active processors. In a hot-add environment, this mask may change during runtime. This parameter is optional and can be specified as **NULL** if the caller does not require the mask of active processors.

Return Value

KeQueryActiveProcessorCount returns the number of currently active processors.

Remarks

A device driver calls the **KeQueryActiveProcessorCount** routine to retrieve the number of currently active processors. Device drivers that are built for Windows Vista and later versions of the Windows operating system should not use the **KeNumberProcessors** kernel variable for this purpose.

Callers cannot assume that the **KeQueryActiveProcessorCount** routine maps processors to bits in the returned **KAFFINITY** value consecutively, or that the routine will return the same bitmask each time it is called.

Callers must also be aware that the value returned by **KeQueryActiveProcessorCount** can change at runtime on versions of Windows that support hot-add CPU functionality.

If necessary, register for notification of changes to the number of processors by calling the KeRegisterProcessorChangeCallback routine.

To query just the affinity mask without getting the number of active processors, use KeQueryActiveProcessors.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryActiveProcessorCountEx routine, which specifies a processor group, instead of **KeQueryActiveProcessorCount**, which does not. However, the implementation of **KeQueryActiveProcessorCount** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryActiveProcessorCount** returns the number of active logical processors in group 0, and writes an affinity mask to **ActiveProcessors* that specifies the set of active logical processors in group 0.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryActiveProcessors

KeRegisterProcessorChangeCallback

KeQueryActiveProcessorCountEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessorCountEx** routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.

Syntax

```
ULONG KeQueryActiveProcessorCountEx(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

The group number. If a multiprocessor system contains *n* groups, valid group numbers range from 0 to *n*-1. To count all active processors in all groups in the system, set this parameter to ALL_PROCESSOR_GROUPS, which is defined in header files Winnt.h and Ntdef.h.

Return Value

KeQueryActiveProcessorCountEx returns the number of active logical processors in the group. If *GroupNumber* is not a valid group number and is not ALL_PROCESSOR_GROUPS, it returns zero.

Remarks

An active logical processor is a logical processor that Windows has started up and added to a multiprocessor system. The term active processor applies to a processor that is available to perform processing work, regardless of whether the processor is currently performing processing work or is idle. In some systems, the number of active processors might remain unchanged from system startup to shutdown. In other systems, Windows might dynamically add active processors while the system is running. Windows never removes active processors from a system. Thus, the number of active processors in a multiprocessor system can increase between system startup and shutdown, but this number never decreases.

A related routine, KeQueryActiveProcessorCount, returns an active processor count, but this routine, unlike **KeQueryActiveProcessorCountEx**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryActiveProcessorCount** returns the active processor count in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryActiveProcessorCount** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryActiveProcessorCountEx** instead of **KeQueryActiveProcessorCount**.

For a code example that uses **KeQueryActiveProcessorCountEx**, see KeGetProcessorNumberFromIndex.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorNumberFromIndex

KeQueryActiveProcessorCount

KeQueryActiveProcessors function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessors** routine returns a bitmask of the currently active processors.

Syntax

```
KAFFINITY KeQueryActiveProcessors(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryActiveProcessors returns a **KAFFINITY** value that represents the set of currently active processors.

Remarks

Callers cannot assume that **KeQueryActiveProcessors** maps processors to bits consecutively, or that the routine consistently uses the same mapping each time it is called. The only valid use for the return value is to determine the number of active processors by counting the number of bits that are set.

Callers must also be aware that the value returned by **KeQueryActiveProcessors** can change during runtime on versions of Windows that support hot-add CPU functionality.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryGroupAffinity routine, which specifies a processor group, instead of **KeQueryActiveProcessors**, which does not. However, the implementation of **KeQueryActiveProcessors** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryActiveProcessors** returns an affinity mask that specifies the set of active logical processors in group 0.

In Windows Vista and later versions of Windows, this routine can be called at any IRQL. However, in Windows Server 2003 and earlier versions of Windows, this routine must be called at IRQL <= APC_LEVEL.

The **KeNumberProcessors** kernel variable is obsolete in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of Windows. **KeNumberProcessors** does not appear in WDK headers for WDK releases starting with Windows Vista SP1; however, the variable is still exported from the kernel, so drivers built for earlier platforms will not break

Windows Server 2008 includes support for Dynamic Hardware Partitioning (DHP) in the Windows Datacenter and Enterprise Edition SKUs. As part of DHP, Windows Server 2008 supports hot adding CPUs at runtime. In a hot-add CPU environment, the number of processors may not remain constant during runtime.

Accordingly, in Windows Server 2008, code that can determine the number of processors must use **KeQueryActiveProcessors** instead of direct references to the kernel variable, **KeNumberProcessors**.

Review any code that currently references **KeNumberProcessors** to make sure that it accommodates changes to

CPU count in hot-add CPU environments.

You can use the PNPCPU tool to simulate hot adding a CPU for testing purposes.

C++

```
#if (NTDDI_VERSION >= NTDDI_VISTA)
extern NTSYSAPI volatile CCHAR KeNumberProcessors;
#else
#if (NTDDI_VERSION >= NTDDI_WINXP)
extern NTSYSAPI CCHAR KeNumberProcessors;
#else
extern PCCHAR KeNumberProcessors;
#endif
#endif
```

Starting with Windows XP, **KeNumberProcessors** is an 8-bit integer value that indicates the number of processors in the platform. In earlier versions of Windows, **KeNumberProcessors** is a pointer to an 8-bit integer value that indicates the number of processors in the platform.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlKeApcLte1, HwStorPortProhibitedDDIs

See Also

KeQueryActiveProcessorCount

KeQueryGroupAffinity

KeQueryGroupAffinity function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryGroupAffinity** routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.

Syntax

```
KAFFINITY KeQueryGroupAffinity(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

The group number. If a multiprocessor system contains *n* groups, the groups are numbered from 0 to *n*-1. To obtain the number of active groups in the system, call the KeQueryActiveGroupCount routine.

Return Value

KeQueryGroupAffinity returns a **KAFFINITY** value that identifies the active logical processors in the specified group. If the *GroupNumber* parameter value is not a valid group number, the routine returns zero.

Remarks

A related routine, KeQueryActiveProcessors, returns an affinity mask that represents a set of active processors, but this routine, unlike **KeQueryGroupAffinity**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryActiveProcessors** returns an affinity mask for the active processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryActiveProcessors** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryGroupAffinity** instead of **KeQueryActiveProcessors**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level

See Also

KAFFINITY

KeQueryActiveProcessors

KeQueryHardwareCounterConfiguration function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryHardwareCounterConfiguration** routine queries the operating system for the list of hardware counters to use for thread profiling.

Syntax

```
NTSTATUS KeQueryHardwareCounterConfiguration(
  PHARDWARE_COUNTER CounterArray,
  ULONG             MaximumCount,
  PULONG            Count
);
```

Parameters

`CounterArray`

A pointer to a caller-allocated buffer into which the routine writes an array of elements of type HARDWARE_COUNTER. Each array element is a structure that contains information about a hardware counter. The array contains one element for each hardware counter that is assigned to thread profiling. If the routine fails, it writes nothing to this buffer.

`MaximumCount`

Specifies the maximum number of elements that the routine can write to the buffer that is pointed to by the *CounterArray* parameter. The size of the caller-allocated buffer must be at least *MaximumCount* * **sizeof**(**HARDWARE_COUNTER**) bytes.

`Count`

A pointer to a location into which the routine writes the number of array elements that it has written to the buffer that is pointed to by the *CounterArray* parameter. If the buffer length that is specified by *MaximumCount* is not large enough to contain the entire array, the routine writes the required length to **Count* and returns STATUS_BUFFER_TOO_SMALL.

Return Value

KeQueryHardwareCounterConfiguration returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The *MaximumCount* parameter specifies a buffer length that is not large enough to contain the counter configuration information.

STA **TUS** **_NO** **T_I** **MP** **LEM** **ENT** **ED**	This routine is not implemented for the processor architecture that the caller is running on.

Remarks

In Windows 7, this routine is implemented only for the x86-based, x64-based, and Itanium-based architectures. If the caller is running on a processor architecture that is not supported, the routine returns STATUS_NOT_IMPLEMENTED.

To set the hardware counter configuration to use for thread profiling, call the KeSetHardwareCounterConfiguration routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

HARDWARE_COUNTER

KeSetHardwareCounterConfiguration

KeQueryHighestNodeNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryHighestNodeNumber** routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.

Syntax

```
USHORT KeQueryHighestNodeNumber(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryHighestNodeNumber returns the highest node number.

Remarks

In a non-uniform memory access (NUMA) multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

If a NUMA multiprocessor system contains *n* nodes, the nodes are numbered from 0 to *n*-1, and **KeQueryHighestNodeNumber** returns *n*-1. If a system contains one node, **KeQueryHighestNodeNumber** returns zero.

If a system does not have a NUMA architecture, **KeQueryHighestNodeNumber** returns zero. An example of non-NUMA architecture is a symmetric multiprocessor system (SMP).

When Windows initializes a NUMA multiprocessor system, it assigns logical processors to nodes and counts the number of resulting nodes. Windows defines the capacity of a node as the number of processors that are present when the system starts in addition to any other logical processors that can be added to the node while the system is running. If, during initialization, Windows encounters a NUMA hardware node that contains more logical processors than will fit into a group, Windows splits the node into smaller, logical nodes. Each of these nodes does not exceed the maximum group size. The value that is returned by **KeQueryHighestNodeNumber** indicates the number of logical nodes in the system.

After the system has been initialized, the node count remains fixed while the system continues to run. This count might include memory-only nodes, which are nodes that contain memory but no active logical processors.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.

Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

KeQueryLogicalProcessorRelationship function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryLogicalProcessorRelationship** routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.

Syntax

```
NTSTATUS KeQueryLogicalProcessorRelationship(
  PPROCESSOR_NUMBER                       ProcessorNumber,
  LOGICAL_PROCESSOR_RELATIONSHIP          RelationshipType,
  PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX Information,
  PULONG                                  Length
);
```

Parameters

`ProcessorNumber`

A pointer to a PROCESSOR_NUMBER structure that identifies the logical processor for which the caller requests relationship information. To request information about all logical processors in the system, set this parameter to **NULL**.

`RelationshipType`

Specifies the type of relationship information that is requested by the caller. Set this parameter to one of the following LOGICAL_PROCESSOR_RELATIONSHIP enumeration values:

- **RelationProcessorCore**
- **RelationNumaNode**
- **RelationCache**
- **RelationProcessorPackage**
- **RelationGroup**
- **RelationAll**

`Information`

A pointer to a caller-allocated buffer into which the routine writes an array of one or more SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structures that contain the information requested by the caller. If the function fails, the contents of this buffer are undefined. Set *Information* = **NULL** to obtain the required buffer length before you allocate the buffer. For more information, see the following Remarks section.

`Length`

A pointer to a location that contains the size, in bytes, of the buffer that is pointed to by *Information*. On entry, * *Length* contains the size of the caller-allocated buffer that is pointed to by *Information*. During the call, the routine overwrites the value that is pointed to by *Length* with the buffer size that is required to contain the requested relationship information.

Return Value

KeQueryLogicalProcessorRelationship returns STATUS_SUCCESS if the call is successful. Possible error return

values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ProcessorNumber* parameter points to a PROCESSOR_NUMBER structure that contains invalid information.
STATUS_INFO_LENGTH_MISMATCH	The caller-allocated buffer that is pointed to by the *Information* parameter is not large enough to contain the requested relationship information.

Remarks

To determine the buffer size to allocate, initially call **KeQueryLogicalProcessorRelationship** with *Information* = **NULL** and **Length* = 0. In response, the routine writes the required buffer size to **Length* and returns STATUS_INFO_LENGTH_MISMATCH. Next, allocate a buffer of the required size and call **KeQueryLogicalProcessorRelationship** a second time. In this second call, set *Information* to the buffer address and **Length* to the buffer size. If the second call succeeds, the routine writes the requested relationship information to the buffer and returns STATUS_SUCCESS.

Examples

The following code example gets processor relationship information for all the logical processors in a multiprocessor system:

```
//
// Get required buffer size.
//
NTSTATUS Status;
PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX Info = NULL;
ULONG BufferSize = 0;
Status = KeQueryLogicalProcessorRelationship(NULL, RelationAll, NULL, &BufferSize);

NT_ASSERT(Status == STATUS_INFO_LENGTH_MISMATCH && BufferSize > 0);

//
// Allocate buffer (assume IRQL <= APC_LEVEL).
//
Info = ExAllocatePoolWithTag(PagedPool, BufferSize, ' gaT');
if (Info == NULL)
{
    Status = STATUS_INSUFFICIENT_RESOURCES;
}

//
// Get processor relationship information.
//
if (NT_SUCCESS(Status))
{
    Status = KeQueryLogicalProcessorRelationship(NULL, RelationAll, Info, &BufferSize);
}
```

The NT_ASSERT macro is defined in the Wdm.h header file. The NT_SUCCESS macro is defined in the Ntdef.h header file.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

LOGICAL_PROCESSOR_RELATIONSHIP

PROCESSOR_NUMBER

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

KeQueryMaximumGroupCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumGroupCount** routine returns the maximum number of groups in a multiprocessor system.

Syntax

```
USHORT KeQueryMaximumGroupCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryMaximumGroupCount returns the maximum number of groups.

Remarks

The value that is returned by **KeQueryMaximumGroupCount** remains constant during runtime. This value depends on the hardware configuration of the multiprocessor system, but it can never exceed a fixed limit that is set by the Windows operating system.

In Windows 7, the maximum number of groups in a multiprocessor system is four, but this value might change in future versions of Windows. The safest way to determine the maximum number of groups in Windows 7 or a later versions of the Windows operating system is to call **KeQueryMaximumGroupCount**. Kernel-mode drivers that call **KeQueryMaximumGroupCount** will not require code changes if the formula that is used to calculate the maximum number of groups changes in a future version of Windows.

To obtain the number of active groups in a multiprocessor system, call the KeQueryActiveGroupCount routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveGroupCount

KeQueryMaximumProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumProcessorCount** routine returns the maximum number of processors.

Syntax

```
ULONG KeQueryMaximumProcessorCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryMaximumProcessorCount returns the maximum number of processors as a ULONG value.

Remarks

The value returned by the **KeQueryMaximumProcessorCount** routine does not change at runtime.

If your code uses an array of buffers, one buffer for each processor, you must decide whether to have a statically sized array based on **KeQueryMaximumProcessorCount** or a dynamically sized array based on KeQueryActiveProcessorCount.

To optimize based on the number of processors, you need a resizable structure for Windows Server 2008. In this case, use **KeQueryActiveProcessorCount**.

If you are not optimizing and if the data structures that result from using maximum processor count are relatively small, a resizable structure is not necessary. In this case, use **KeQueryMaximumProcessorCount** to determine size for a static array.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryMaximumProcessorCountEx routine, which specifies a processor group, instead of **KeQueryMaximumProcessorCount**, which does not. However, the implementation of **KeQueryMaximumProcessorCount** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryMaximumProcessorCount** returns the maximum number of logical processors that can be in group 0.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal

Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCount

KeQueryMaximumProcessorCountEx

KeQueryMaximumProcessorCountEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumProcessorCountEx** routine returns the maximum number of logical processors in a specified group in a multiprocessor system.

Syntax

```
ULONG KeQueryMaximumProcessorCountEx(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

Specifies a group number. If a multiprocessor system contains *n* groups, the groups are numbered from 0 to *n*-1. To obtain the maximum number of processors in the system, set this parameter to ALL_PROCESSOR_GROUPS, which is defined in the Winnt.h and Ntdef.h header files.

Return Value

KeQueryMaximumProcessorCountEx returns the maximum number of logical processors in the specified group or in the system, as indicated by the value of the *GroupNumber* parameter.

Remarks

The value that is returned by **KeQueryMaximumProcessorCountEx** remains constant during runtime. This value depends on the hardware configuration of the multiprocessor system, but it can never exceed a fixed limit that is set by the Windows operating system. This limit is 32 logical processors for 32-bit versions of Windows and 64 logical processors for 64-bit versions of Windows.

In contrast, the value that is returned by the KeQueryActiveProcessorCountEx routine can change during runtime as processors are dynamically added to the system.

A related routine, KeQueryMaximumProcessorCount, returns a maximum processor count, but this routine, unlike **KeQueryMaximumProcessorCountEx**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryMaximumProcessorCount** returns the maximum number of logical processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryMaximumProcessorCount** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryMaximumProcessorCountEx** instead of **KeQueryMaximumProcessorCount**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryMaximumProcessorCount

KeQueryNodeActiveAffinity function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryNodeActiveAffinity** routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.

Syntax

```
void KeQueryNodeActiveAffinity(
  USHORT          NodeNumber,
  PGROUP_AFFINITY Affinity,
  PUSHORT         Count
);
```

Parameters

`NodeNumber`

The node number. If a multiprocessor system contains *n* nodes, the nodes are numbered from 0 to *n*-1. To obtain the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

`Affinity`

A pointer to a caller-allocated buffer into which the routine writes a GROUP_AFFINITY structure. This structure contains the group number of the group that contains the node that is identified by *NodeNumber*, and an affinity mask that indicates which logical processors in the node are active. You can set this parameter to **NULL** if you do not need this information.

`Count`

A pointer to a location into which the routine writes the number of active processors that are represented in the node affinity mask that is pointed to by *Affinity*. You can set this parameter to **NULL** if you do not need this information.

Return Value

None

Remarks

The caller-allocated buffer that is pointed to by the *Affinity* parameter must be large enough to contain a **GROUP_AFFINITY** structure. The **Mask** member of this structure contains an affinity mask that indicates which processors are active. If a processor is active, the corresponding bit in the mask is one. All other bits are zero.

In a NUMA multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

The number of processors in a node cannot exceed the number of bits in the affinity mask in the structure that is pointed to by *Affinity*. The affinity mask also determines the maximum number of processors in a group.

If, during system initialization, Windows encounters a NUMA hardware node that contains more logical processors

than will fit into a group, Windows splits the node into smaller, logical nodes. Each of these nodes does not exceed the maximum group size. The *NodeNumber* parameter identifies a logical node. To obtain the maximum number of processors per group, call the KeQueryMaximumProcessorCountEx routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

GROUP_AFFINITY

KeQueryHighestNodeNumber

KeQueryMaximumProcessorCountEx

KeQueryNodeMaximumProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryNodeMaximumProcessorCount** routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.

Syntax

```
USHORT KeQueryNodeMaximumProcessorCount(
  USHORT NodeNumber
);
```

Parameters

`NodeNumber`

The node number. If a NUMA multiprocessor system contains *n* nodes, valid node numbers are in the range 0 to *n*-1. To get the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

Return Value

KeQueryNodeMaximumProcessorCount returns the maximum number of logical processors.

Remarks

In a multiprocessor system with a NUMA architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

The count returned by this routine includes any logical processors that can be dynamically added to the node while the multiprocessor system is running.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryHighestNodeNumber

KeQueryTickCount macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryTickCount** routine maintains a count of the interval timer interrupts that have occurred since the system was booted.

Syntax

```
void KeQueryTickCount(
   CurrentCount
);
```

Parameters

`CurrentCount`

Pointer to the tick count value on return from **KeQueryTickCount**.

Return Value

None

Remarks

The *TickCount* value increases by one at each interval timer interrupt while the system is running.

The preferred method of determining elapsed time is by using *TickCount* for relative timing and time stamps.

To determine the absolute elapsed time multiply the returned *TickCount* by the KeQueryTimeIncrement return value using compiler support for 64-bit integer operations.

You should not make any assumptions about the length of a tick, because it might vary depending on hardware and other considerations.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryInterruptTime

KeQueryPerformanceCounter

KeQueryTimeIncrement

KeRaiseIrqlToDpcLevel function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRaiseIrqlToDpcLevel** routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

Syntax

```
_DECL_HAL_KE_IMPORT KIRQL KeRaiseIrqlToDpcLevel(

);
```

Parameters

This function has no parameters.

Return Value

KeRaiseIrqlToDpcLevel returns the IRQL at which the call occurred.

Remarks

Any caller of **KeRaiseIrqlToDpcLevel** should save the returned IRQL value. Every such caller must restore the original IRQL as quickly as possible by passing this returned IRQL in a subsequent call to KeLowerIrql.

Callers of **KeRaiseIrqlToDpcLevel** must be running at IRQL <= DISPATCH_LEVEL. Otherwise, a call to this routine causes a bug check.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	Hal.lib
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeGetCurrentIrql

KeLowerIrql

KeRaiseIrql

KeSetBasePriorityThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetBasePriorityThread** routine sets the run-time priority, relative to the current process, for a given thread.

Syntax

```
LONG KeSetBasePriorityThread(
  PKTHREAD Thread,
  LONG     Increment
);
```

Parameters

`Thread`

Pointer to a dispatcher object of type KTHREAD.

`Increment`

Specifies the value to be added to the base priority of the process for the *Thread*.

Return Value

KeSetBasePriorityThread returns the previous base priority increment of the given thread. The previous base priority increment is defined as the difference between the specified thread's old base priority and the base priority of the thread's process.

Remarks

The new base priority is computed by adding the given *Increment*, which can be a negative value, to the base priority of the specified thread's process. The resultant value is stored as the base priority of the specified thread.

Drivers that set up device-dedicated threads with variable priority attributes can call this routine to set such a thread's priority relative to the system process in which the thread is created.

The new base priority is restricted to the priority class of the given thread's process. Therefore, the base priority is not allowed to cross over from a variable priority class to a real-time priority class or vice versa.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

KeGetCurrentThread

KeQueryPriorityThread

KeSetPriorityThread

KeSetHardwareCounterConfiguration function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetHardwareCounterConfiguration** routine specifies a list of hardware counters to use for thread profiling.

Syntax

```
NTSTATUS KeSetHardwareCounterConfiguration(
  PHARDWARE_COUNTER CounterArray,
  ULONG             Count
);
```

Parameters

`CounterArray`

A pointer to a HARDWARE_COUNTER array that describes the hardware counter configuration to use for thread profiling. Each array element is a structure that describes a hardware counter. Before the routine returns, it copies the contents of this array into its internal data structures.

`Count`

Specifies the number of elements in the array that is pointed to by the *CounterArray* parameter.

Return Value

KeSetHardwareCounterConfiguration returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of the *Count* parameter exceeds the maximum number of counters that is specified by the MAX_HW_COUNTERS constant, which is defined in the Ntddk.h header file.

STATUS_WMI_ALREADY_ENABLED	One or more of the counters that are specified in the *CounterArray* array are already enabled.
STATUS_NOT_IMPLEMENTED	This routine is not implemented for the processor architecture that the caller is running on.

Remarks

In Windows 7, this routine is implemented only for the x86-based, x64-based, and Itanium-based architectures. If a caller is running on a processor architecture that is not supported, the routine returns STATUS_NOT_IMPLEMENTED.

This routine tells the operating system which hardware counters to use for thread profiling. Call this routine only when thread profiling is disabled. If the *CounterArray* array specifies any hardware counters that are currently being used, the routine fails and returns STATUS_WMI_ALREADY_ENABLED.

An application thread can enable thread profiling to obtain a set of performance measurements from the hardware counters in the performance monitoring unit (PMU) of the local processor. The operating system supports only one profiling application at a time. Concurrent instances of a thread-profiling application are not supported. A thread can enable thread profiling for itself but not for other threads.

When thread profiling is enabled, the operating system uses the hardware counters that were specified in the last call to **KeSetHardwareCounterConfiguration**. Each successful **KeSetHardwareCounterConfiguration** call replaces any hardware counter configuration that might have been set in a previous **KeSetHardwareCounterConfiguration** call.

Set *Count* = 0 to specify an empty hardware counter configuration. This configuration effectively prevents the use of hardware counters for thread profiling. The default hardware counter configuration that exists after system startup and before the initial **KeSetHardwareCounterConfiguration** call is an empty configuration.

The effect of a successful **KeSetHardwareCounterConfiguration** call is global. If a thread in any process is profiled, the profiler uses the hardware counter configuration that was set by the last call to **KeSetHardwareCounterConfiguration**. In a multiprocessor system, a **KeSetHardwareCounterConfiguration** call sets the hardware counter configuration to use for thread profiling across all processors in the system, although each processor uses its own set of hardware counters.

To avoid resource conflicts, all drivers that use counter resources should use the HalAllocateHardwareCounters and HalFreeHardwareCounters routines to coordinate their sharing of these resources.

To query the operating system for the hardware counter configuration that is currently in effect for thread profiling, call the KeQueryHardwareCounterConfiguration routine.

Virtualization software typically does not virtualize hardware performance counters. Thus, hardware performance counters are unlikely to be available in a virtual machine.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

HARDWARE_COUNTER

HalAllocateHardwareCounters

HalFreeHardwareCounters

KeQueryHardwareCounterConfiguration

KeSetImportanceDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetImportanceDpc** routine specifies how soon the DPC routine is run.

Syntax

```
void KeSetImportanceDpc(
  PRKDPC          Dpc,
  KDPC_IMPORTANCE Importance
);
```

Parameters

`Dpc`

Pointer to the caller's DPC object, which KeInitializeDpc already initialized.

`Importance`

Specifies one of the following system-defined values to determine the behavior of KeInsertQueueDpc and IoRequestDpc when either routine is used to queue the DPC.

LowImportance

Place the DPC at the end of the DPC queue, and do not begin processing of the queue.

MediumImportance

Place the DPC at the end of the DPC queue. If the DPC is assigned to the current processor's DPC queue, begin processing the queue immediately. **MediumImportance** is the default value for *Importance*.

MediumHighImportance

Place the DPC at the end of the DPC queue, and begin processing the queue immediately. MediumHighImportance is available only on Windows Vista and later versions of Windows.

HighImportance

Place the DPC at the beginning of the DPC queue, and begin processing the queue immediately.

Return Value

None

Remarks

The **KeSetImportanceDpc** routine influences how soon a DPC is run after it is queued by determining:

- The location of the DPC within the DPC queue. Typically, the **KeInsertQueueDpc** and **IoRequestDpc** routines place a DPC at the end of the queue. If a driver first calls **KeSetImportanceDpc** with *Importance* = **HighImportance**, **KeInsertQueueDpc** and **IoRequestDpc** will place the DPC at the beginning of the queue.
- When the system begins processing the DPC queue. Typically, **KeInsertQueueDpc** and **IoRequestDpc** immediately begin processing the DPC queue for the current processor. Drivers can specify different values for *Importance* to change this behavior.

By default, DPCs are assigned to the DPC queue for the current processor, so specifying **MediumImportance** or **MediumHighImportance** for *Importance* has the same effect. However, drivers can use KeSetTargetProcessorDpc to change the processor that the DPC will be assigned to.

For Windows Vista and later versions of the Windows operating system, you can use **KeSetImportanceDpc** for threaded DPCs. If the caller sets *Importance* to **HighImportance**, the DPC is placed at the beginning of the queue; otherwise, it is placed at the end. The routine does not affect when the threaded DPC queue is processed. Threaded DPCs are always processed by a dedicated thread at IRQL = PASSIVE_LEVEL. For more information about threaded DPCs, see Threaded DPCs.

Note that a driver must call **KeSetImportanceDpc** before it calls **KeInsertQueueDpc** and **IoRequestDpc** to have any effect.

For more information about how the system processes the DPC queue, see Organization of DPC Queues.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoRequestDpc

KeInitializeDpc

KeInsertQueueDpc

KeSetTargetProcessorDpc

KeSynchronizeExecution

KeSetTargetProcessorDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetTargetProcessorDpc** routine specifies the processor that a DPC routine will be run on.

Syntax

```
void KeSetTargetProcessorDpc(
  PRKDPC Dpc,
  CCHAR  Number
);
```

Parameters

`Dpc`

Pointer to the caller's DPC object, which KeInitializeDpc already initialized.

`Number`

Specifies the zero-based number of the target processor on which the DPC should be queued and executed.

Return Value

None

Remarks

On multiprocessor systems, each processor has its own DPC queue. The **KeSetTargetProcessorDpc** routine specifies which processor's queue the system should use when the driver calls KeInsertQueueDpc or IoRequestDpc to queue a DPC to be run later.

Starting with Windows Vista, you can also use **KeSetTargetProcessorDpc** to specify the target processor for threaded DPCs.

A call to **KeSetTargetProcessorDpcEx** that occurs after a DPC object has been queued has no effect on the selection of a processor for the DPC routine to run on. To control the selection of the target processor, a **KeSetTargetProcessorDpc** call must occur before the call to **KeInsertQueueDpc** or **IoRequestDpc** that queues the DPC object.

For more information about DPC queues, see Organization of DPC Queues.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeSetTargetProcessorDpcEx routine, which specifies a processor group, instead of **KeSetTargetProcessorDpc**, which does not. However, the implementation of **KeSetTargetProcessorDpc** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, if *Number* is less than the number of active logical processors in group 0, **KeSetTargetProcessorDpc** sets the target for the DPC to the processor in group 0 that is specified by *Number*. Otherwise, the DPC target does not change.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoRequestDpc

KeGetCurrentProcessorNumber

KeInitializeDpc

KeInsertQueueDpc

KeSetImportanceDpc

KeSetTargetProcessorDpcEx

MmAllocateContiguousMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmAllocateContiguousMemory** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemory(
  SIZE_T           NumberOfBytes,
  PHYSICAL_ADDRESS HighestAcceptableAddress
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

Return Value

MmAllocateContiguousMemory returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

MmAllocateContiguousMemory allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemory** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemory** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemory**.

Note When physical memory is fragmented on a computer that has a large amount of RAM, calls to **MmAllocateContiguousMemory**, which require the operating system to search for contiguous blocks of memory, can severely degrade performance. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemory**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

Memory that **MmAllocateContiguousMemory** allocates is uninitialized. A kernel-mode driver must first set this memory to zero if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

AllocateCommonBuffer

DriverEntry

MmAllocateContiguousMemorySpecifyCache

MmAllocateNonCachedMemory

MmFreeContiguousMemory

MmAllocateContiguousMemorySpecifyCache function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocateContiguousMemorySpecifyCache** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemorySpecifyCache(
  SIZE_T              NumberOfBytes,
  PHYSICAL_ADDRESS    LowestAcceptableAddress,
  PHYSICAL_ADDRESS    HighestAcceptableAddress,
  PHYSICAL_ADDRESS    BoundaryAddressMultiple,
  MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching allowed for the requested memory.

Return Value

MmAllocateContiguousMemorySpecifyCache returns the base virtual address for the allocated memory. If the system is unable to allocate the requested buffer, the routine returns **NULL**.

Remarks

MmAllocateContiguousMemorySpecifyCache allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemorySpecifyCache** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemorySpecifyCache** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemorySpecifyCache**.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** If you use the **MmAllocateContiguousMemorySpecifyCache** routine on computers with large amounts of memory, the operating system's performance might severely degrade when the system tries to create a contiguous chunk of memory. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemorySpecifyCache**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

> **Note** Memory that **MmAllocateContiguousMemorySpecifyCache** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemory

MmFreeContiguousMemory

MmAllocateContiguousMemorySpecifyCacheNode function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocateContiguousMemorySpecifyCacheNode** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemorySpecifyCacheNode(
  SIZE_T              NumberOfBytes,
  PHYSICAL_ADDRESS    LowestAcceptableAddress,
  PHYSICAL_ADDRESS    HighestAcceptableAddress,
  PHYSICAL_ADDRESS    BoundaryAddressMultiple,
  MEMORY_CACHING_TYPE CacheType,
  NODE_REQUIREMENT    PreferredNode
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching requested for the contiguous physical memory.

`PreferredNode`

The preferred node number. If a multiprocessor system contains N nodes, the nodes are numbered 0 to N-1. If MM_ANY_NODE_OK is specified or the machine only has one node, then the allocation is satisfied from any node. Otherwise, the allocation is made from the preferred node or if a satisfactory range cannot be found from the preferred node, then **NULL** is returned.

Return Value

MmAllocateContiguousMemorySpecifyCacheNode returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

A kernel-mode device driver calls this routine to allocate a contiguous block of physical memory. In a non-uniform memory access (NUMA) multiprocessor system, the caller can specify a preferred node from which to allocate the memory. A node is a collection of processors that share fast access to a region of memory. In a non-NUMA multiprocessor or a single-processor system, **MmAllocateContiguousMemorySpecifyCacheNode** treats all memory as belonging to a single node and allocates memory from this node.

MmAllocateContiguousMemorySpecifyCacheNode allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemorySpecifyCacheNode** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemorySpecifyCacheNode** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemorySpecifyCacheNode**.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** If you use the **MmAllocateContiguousMemorySpecifyCacheNode** routine on computers with large amounts of memory, the operating system's performance might severely degrade when the system tries to create a contiguous chunk of memory. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemorySpecifyCacheNode**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

> **Note** Memory that **MmAllocateContiguousMemorySpecifyCacheNode** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows Vista.

Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DriverEntry

MEMORY_CACHING_TYPE

MmFreeContiguousMemory

MmAllocateContiguousNodeMemory function

1/25/2019 • 4 minutes to read • Edit Online

The **MmAllocateContiguousNodeMemory** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousNodeMemory(
  SIZE_T           NumberOfBytes,
  PHYSICAL_ADDRESS LowestAcceptableAddress,
  PHYSICAL_ADDRESS HighestAcceptableAddress,
  PHYSICAL_ADDRESS BoundaryAddressMultiple,
  ULONG            Protect,
  NODE_REQUIREMENT PreferredNode
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`Protect`

Flag bits that specify the protection to use for the allocated memory. The caller must set one (but not both) of the following flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_READWRITE	Allocate read/write, no-execute (NX) memory. Most callers should set this flag bit. For more information, see Remarks.

PAGE_EXECUTE_READWRITE	Allocate read/write memory that is executable. This flag bit should be set only if the caller requires the ability to execute instructions in the allocated memory.

In addition, the caller can set one (but not both) of the following optional flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_NOCACHE	Allocate non-cached memory. This flag bit is similar in effect to calling MmAllocateContiguousMemorySpecifyCache with *CacheType* set to **MmNonCached**.
PAGE_WRITECOMBINE	Allocate write-combined memory. This flag bit is similar in effect to calling **MmAllocateContiguousMemorySpecifyCache** with *CacheType* set to **MmWriteCombined**.

If neither PAGE_NOCACHE nor PAGE_WRITECOMBINE is specified, the allocated memory is fully cached. In this case, the effect is similar to calling **MmAllocateContiguousMemorySpecifyCache** with *CacheType* set to **MmCached**.

`PreferredNode`

The preferred node number. If a multiprocessor system contains N nodes, the nodes are numbered from 0 to N-1. If the caller sets *PreferredNode* to MM_ANY_NODE_OK, the routine chooses which node to allocate memory from. Otherwise, if memory in the specified address range cannot be allocated from the preferred node, the routine returns **NULL**.

Return Value

MmAllocateContiguousNodeMemory returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

A kernel-mode device driver calls this routine to allocate a contiguous block of physical memory. The calling driver can specify whether to use no-execute (NX) memory for the allocation. In a non-uniform memory access (NUMA) multiprocessor system, the caller can specify a preferred node from which to allocate the memory. A node is a collection of processors that share fast access to a region of memory. In a non-NUMA multiprocessor or a single-processor system, **MmAllocateContiguousNodeMemory** treats all memory as belonging to a single node and allocates memory from this node.

MmAllocateContiguousNodeMemory allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousNodeMemory** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousNodeMemory** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousNodeMemory**.

MmAllocateContiguousNodeMemory is similar to the MmAllocateContiguousMemorySpecifyCacheNode routine. Unlike **MmAllocateContiguousMemorySpecifyCacheNode**, **MmAllocateContiguousNodeMemory** can be used to allocate no-execute (NX) memory. As a best practice, a driver should allocate NX memory unless the driver explicitly requires the ability to execute instructions in the allocated memory. By allocating NX memory, a driver improves security by preventing malicious software from executing instructions in this memory. Memory allocated by the MmAllocateContiguousMemory, MmAllocateContiguousMemorySpecifyCache, and **MmAllocateContiguousMemorySpecifyCacheNode** routines is always executable.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** Memory that **MmAllocateContiguousNodeMemory** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DriverEntry

MmAllocateContiguousMemory

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemory

MmAllocateNonCachedMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmAllocateNonCachedMemory** routine allocates a virtual address range of noncached and cache-aligned memory.

Syntax

```
PVOID MmAllocateNonCachedMemory(
  SIZE_T NumberOfBytes
);
```

Parameters

`NumberOfBytes`

Specifies the size in bytes of the range to be allocated.

Return Value

If the requested memory cannot be allocated, the return value is **NULL**. Otherwise, it is the base virtual address of the allocated range.

Remarks

MmAllocateNonCachedMemory can be called from a DriverEntry routine to allocate a noncached block of virtual memory for various device-specific buffers. The function always returns a full multiple of the virtual memory page size, of nonpaged system-address-space memory, regardless of the requested allocation size.

Noncached allocations are aligned on an integral multiple of the processor's data-cache-line size to prevent cache and coherency problems.

The physical memory pages that **MmAllocateNonCachedMemory** returns are typically not contiguous pages.

> **Note** Memory that **MmAllocateNonCachedMemory** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

AllocateCommonBuffer

KeGetDcacheFillSize

MmAllocateContiguousMemory

MmAllocateContiguousMemorySpecifyCache

MmFreeNonCachedMemory

MmCopyMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmCopyMemory** routine copies the specified range of virtual or physical memory into the caller-supplied buffer.

Syntax

```
NTSTATUS MmCopyMemory(
  PVOID           TargetAddress,
  MM_COPY_ADDRESS SourceAddress,
  SIZE_T          NumberOfBytes,
  ULONG           Flags,
  PSIZE_T         NumberOfBytesTransferred
);
```

Parameters

`TargetAddress`

A pointer to a caller-supplied buffer. This buffer must be in nonpageable memory.

`SourceAddress`

An MM_COPY_ADDRESS structure, passed by value, that contains either the virtual address or the physical address of the data to be copied to the buffer pointed to by *TargetAddress*.

`NumberOfBytes`

The number of bytes to copy from *SourceAddress* to *TargetAddress*.

`Flags`

Flags that indicate whether *SourceAddress* is a virtual address or a physical address. The following flag bits are defined for this parameter.

FLAG BIT	DESCRIPTION
MM_COPY_MEMORY_PHYSICAL	*SourceAddress* specifies a physical address.
MM_COPY_MEMORY_VIRTUAL	*SourceAddress* specifies a virtual address.

These two flag bits are mutually exclusive. The caller must set one or the other, but not both.

`NumberOfBytesTransferred`

A pointer to a location to which the routine writes the number of bytes successfully copied from the *SourceAddress* location to the buffer at *TargetAddress*.

Return Value

MmCopyMemory returns STATUS_SUCCESS if the entire range has been copied successfully. Otherwise, an error status is returned and the caller must inspect the output value pointed to by the *NumberOfBytesTransferred*

parameter to determine how many bytes were actually copied.

Remarks

Kernel-mode drivers can call this routine to safely access arbitrary physical or virtual addresses.

If the MM_COPY_MEMORY_PHYSICAL flag is set, *SourceAddress* should point to regular memory that is under control of the operating system. **MmCopyMemory** will return an error status code for physical addresses that refer to I/O space, which includes memory-mapped devices and firmware tables. To access physical memory in I/O space, drivers can use the MmMapIoSpace routine.

If the MM_COPY_MEMORY_VIRTUAL flag is set, *SourceAddress* can point to either a buffer in system address space, or a buffer in the user address space of the current process. If the caller does not control the lifetime of the allocation containing the specified source address, **MmCopyMemory** might fail or might return inconsistent data, but will not cause a system crash—even for system addresses that are invalid and would trigger a bug check if referenced directly. **MmCopyMemory** will return an error status code for system virtual addresses that refer to I/O space.

If memory at the virtual address specified by *SourceAddress* is not resident, **MmCopyMemory** will try to make it resident.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	ntddk.h
Library	Ntoskrnl.lib
IRQL	"<= APC_LEVEL"

See Also

MM_COPY_ADDRESS

MmMapIoSpace

MmFreeContiguousMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeContiguousMemory** routine releases a range of physically contiguous memory that was allocated by an **MmAllocateContiguousMemory*Xxx*** routine.

Syntax

```
void MmFreeContiguousMemory(
  PVOID BaseAddress
);
```

Parameters

`BaseAddress`

Pointer to the virtual address of the memory to be freed.

Return Value

None

Remarks

The **MmFreeContiguousMemory** routine frees a block of physically contiguous memory that was allocated by a previous call to the MmAllocateContiguousMemory, MmAllocateContiguousMemorySpecifyCache, or MmAllocateContiguousMemorySpecifyCacheNode routine. The *BaseAddress* parameter must be the base address that was obtained from the previous call to the **MmAllocateContiguousMemory*Xxx*** routine.

A device driver that must use contiguous memory should allocate only what it needs during driver initialization because physical memory is likely to become fragmented as the system runs. Such a driver must deallocate the memory when the driver is done using the memory.

Callers of **MmFreeContiguousMemory** must be running at IRQL = APC_LEVEL. For Windows Server 2008 and later versions of the Windows operating system, you can also call **MmFreeContiguousMemory** with IRQL <= DISPATCH_LEVEL. However, you can improve driver performance by calling at APC_LEVEL or below.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlMmDispatch, HwStorPortProhibitedDDIs

See Also

MmAllocateContiguousMemory

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemorySpecifyCache function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeContiguousMemorySpecifyCache** routine frees a buffer that was allocated by an **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

Syntax

```
void MmFreeContiguousMemorySpecifyCache(
  PVOID               BaseAddress,
  SIZE_T              NumberOfBytes,
  MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`BaseAddress`

Specifies the base address of the buffer to be freed. Must match the address returned by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** call that allocated the buffer.

`NumberOfBytes`

Specifies the size in bytes of the buffer to be freed. Must match the size requested when the buffer was allocated by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

`CacheType`

Specifies the cache type of the buffer to be freed. Must match the cache type requested when the buffer was allocated by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

Return Value

None

Remarks

The **MmFreeContiguousMemorySpecifyCache** routine frees a block of physically contiguous memory that was allocated by a previous call to the MmAllocateContiguousMemorySpecifyCache or MmAllocateContiguousMemorySpecifyCacheNode routine. However, MmFreeContiguousMemory is the preferred routine to use to free memory that was allocated by an **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine. **MmFreeContiguousMemory** is faster than **MmFreeContiguousMemorySpecifyCache** and requires fewer parameters.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemory

MmFreeNonCachedMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeNonCachedMemory** routine releases a range of noncached memory that was allocated by the MmAllocateNonCachedMemory routine.

Syntax

```
void MmFreeNonCachedMemory(
  PVOID  BaseAddress,
  SIZE_T NumberOfBytes
);
```

Parameters

`BaseAddress`

Pointer to the virtual address of the memory to be freed.

`NumberOfBytes`

Specifies the size of the range to be freed. This value must match the size passed in a preceding call to **MmAllocateNonCachedMemory**.

Return Value

None

Remarks

The **MmFreeNonCachedMemory** routine performs the opposite action of MmAllocateNonCachedMemory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmAllocateNonCachedMemory

MmGetPhysicalAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmGetPhysicalAddress** routine returns the physical address corresponding to a valid nonpaged virtual address.

Syntax

```
PHYSICAL_ADDRESS MmGetPhysicalAddress(
  PVOID BaseAddress
);
```

Parameters

`BaseAddress`

Pointer to the virtual address for which to return the physical address.

Return Value

MmGetPhysicalAddress returns the physical address that corresponds to the given virtual address.

Do not use this routine to obtain physical addresses for use with DMA operations. For information about the proper techniques for performing DMA operations, see Adapter Objects and DMA.

Remarks

Callers of **MmGetPhysicalAddress** can be running at any IRQL, provided that the *BaseAddress* value is valid.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

MmIsAddressValid

MmIsNonPagedSystemAddressValid

MmMapIoSpace

MmProbeAndLockPages

MmIsAddressValid function

1/25/2019 • 2 minutes to read • Edit Online

The **MmIsAddressValid** routine checks whether a page fault will occur for a read or write operation at a given virtual address.

> **Warning** We do not recommend using this function.

Syntax

```
BOOLEAN MmIsAddressValid(
  PVOID VirtualAddress
);
```

Parameters

`VirtualAddress`

A pointer to the nonpaged virtual address to check. The caller must ensure that this address cannot be paged out or deleted for the duration of this call. Even after the return from the call, you must not page out or delete this address. If you do page out or delete this address, the return value might be unreliable. Paging out or deleting this address might cause the computer to stop responding (that is, *crash*).

Return Value

If no page fault would occur from reading or writing at the given virtual address, **MmIsAddressValid** returns **TRUE**.

Remarks

Even if **MmIsAddressValid** returns **TRUE**, accessing the address can cause page faults unless the memory has been locked down or the address is a valid nonpaged pool address.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmProbeAndLockPages

MmIsThisAnNtAsSystem function

1/25/2019 • 2 minutes to read • Edit Online

The **MmIsThisAnNtAsSystem** routine is obsolete for Windows XP and later versions of Windows. Use RtlGetVersion or RtlVerifyVersionInfo instead.

The **MmIsThisAnNtAsSystem** routine checks whether the current platform is running a server version of the NT-based operating system.

Syntax

```
BOOLEAN MmIsThisAnNtAsSystem(

);
```

Parameters

This function has no parameters.

Return Value

If the current platform is a server, **MmIsThisAnNtAsSystem** returns **TRUE**.

Remarks

Drivers can use this routine during initialization, along with **MmQuerySystemSize**, for sizing estimates of how many resources to allocate. For example, if **MmIsThisAnNtAsSystem** returns **TRUE**, the caller can increase the number of threads or the number of initially allocated entries for a lookaside list that it creates in medium and large systems.

Requirements

Minimum supported client	Obsolete for Windows XP and later versions of Windows. Use RtlGetVersion or RtlVerifyVersionInfo instead.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

MmQuerySystemSize

MmLockPagableSectionByHandle function

1/25/2019 • 2 minutes to read • Edit Online

The **MmLockPagableSectionByHandle** routine locks a pageable code or data section into system memory by incrementing the reference count on the handle to the section.

Syntax

```
void MmLockPagableSectionByHandle(
  PVOID ImageSectionHandle
);
```

Parameters

`ImageSectionHandle`

The handle to a pageable code or data section. The caller obtained this handle from a previous call to the MmLockPagableCodeSection or MmLockPagableDataSection routine.

Return Value

None

Remarks

A driver initially calls **MmLockPagableCodeSection** or **MmLockPagableDataSection** to obtain a handle to a pageable code or data section. Thereafter, the driver can call **MmLockPagableSectionByHandle** and MmUnlockPagableImageSection to increment and decrement the reference count on the handle.

The **MmLockPagableCodeSection** or **MmLockPagableDataSection** call locks the pageable section into system memory, returns a handle to the section, and initializes the reference count on the handle to one. **MmLockPagableSectionByHandle** increments the reference count by one, and **MmUnlockPagableImageSection** decrements the reference count by one.

If the reference count on the handle is nonzero, the pageable section is locked. If the reference count is zero, the section is unlocked.

The handle to the pageable section remains valid as long as the driver stays loaded in memory. The driver can use the same handle to repeatedly lock and unlock the section. That is, the driver can reuse a handle whose reference count is zero. **MmLockPagableSectionByHandle** checks to see if the referenced section is resident in the caller's address space and, if it is, simply increments the reference count on the section. If the section is not resident, **MmLockPagableSectionByHandle** pages in the section, locks it in system space, and sets the reference count to one.

To avoid leaking system memory, the reference count on the handle must be zero when the driver is unloaded from memory. Otherwise, the pageable section will stay locked into system memory after the driver unloads.

If the driver is preparing to unload, and the reference count on the handle is nonzero, the driver should call **MmUnlockPagableImageSection** to decrement the count to zero before the unload occurs.

A driver cannot call **MmLockPagableSectionByHandle** to lock down user buffers passed in IRPs. Use

MmProbeAndLockPages instead.

For more information about paging code and data, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmLockPagableCodeSection

MmLockPagableDataSection

MmPageEntireDriver

MmProbeAndLockPages

MmResetDriverPaging

MmUnlockPagableImageSection

MmSecureVirtualMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmSecureVirtualMemory** routine secures a user-space memory address range so that it cannot be freed and its page protection cannot be made more restrictive.

Syntax

```
HANDLE MmSecureVirtualMemory(
  PVOID  Address,
  SIZE_T Size,
  ULONG  ProbeMode
);
```

Parameters

`Address`

The beginning of the user virtual address range to secure.

`Size`

The size, in bytes, of the virtual address range to secure.

`ProbeMode`

The most restrictive page protection that is allowed. Use PAGE_READWRITE to specify that the address range must remain both readable and writable, or use PAGE_READONLY to specify that the address range must only remain readable.

PROBEMODE	MEANING
PAGE_READWRITE	Protection cannot be changed to PAGE_NOACCESS or PAGE_READONLY. All other protection changes are allowed.
PAGE_READONLY	Protection cannot be changed to PAGE_NOACCESS. All other protection changes are allowed.

Return Value

On success, **MmSecureVirtualMemory** returns an opaque pointer value that the driver passes to the MmUnsecureVirtualMemory routine to unsecure the memory address range. If the routine is unable to secure the memory address range, it returns **NULL**.

Remarks

MmSecureVirtualMemory can be used to avoid certain race conditions on user-mode buffers. For example, if a driver checks to see if the buffer is writable, but then the originating user-mode process changes the buffer to be read-only before the driver can write to the buffer, then a race condition can result. The driver can use **MmSecureVirtualMemory** with PAGE_READWRITE probe mode to guarantee that the buffer will remain writable until the driver calls **MmUnsecureVirtualMemory**. The routine also protects against the originating

user-mode process freeing the buffer. Here are a few guidelines about calling these routines:

- Calling **MmSecureVirtualMemory** with PAGE_READONLY does not guarantee that the buffer will remain read-only. The read-only probe mode prevents the user from changing the protection of the buffer to PAGE_NOACCESS. It does *not* prevent changing the protection to PAGE_READWRITE (or PAGE_WRITECOPY, for mapped views).
- If a driver calls **MmSecureVirtualMemory** and does not call MmUnsecureVirtualMemory, the memory is automatically unsecured when the process terminates.
- If the driver calls MmUnsecureVirtualMemory, it must call it in the context of the process in which the memory was originally secured, and before that process terminates.
- Typically drivers need to reference the process when they secure the memory, then later call KeStackAttachProcess to switch to the context of that process before calling MmUnsecureVirtualMemory.
- To detect process termination drivers can use PsSetCreateProcessNotifyRoutine. Alternatively, the process can submit an IRP with a cancel routine that is invoked by the I/O manager when the process is exiting. In the cancel routine the driver can attach to the process and call MmUnsecureVirtualMemory.

While **MmSecureVirtualMemory** can be used to guarantee that reading or writing user memory will not raise an exception due to insufficient page permissions, it does not protect against other types of exceptions. For example, it does not protect against exceptions raised when the system finds a bad disk block in the page file. Therefore, drivers must still wrap all user memory accesses in a **try/except** block. Because of this, we recommend that drivers do not use this function. For more information, see Handling Exceptions.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmUnsecureVirtualMemory

MmUnsecureVirtualMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnsecureVirtualMemory** routine unsecures a memory address range secured by the MmSecureVirtualMemory routine.

Syntax

```
void MmUnsecureVirtualMemory(
  HANDLE SecureHandle
);
```

Parameters

`SecureHandle`

Specifies the value returned by MmSecureVirtualMemory for the memory address range.

Return Value

None

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. This routine is not available in Windows 98/Me.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmSecureVirtualMemory

NtOpenProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenProcess** routine opens a handle to a process object and sets the access rights to this object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcess(
  PHANDLE            ProcessHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PCLIENT_ID         ClientId
);
```

Parameters

`ProcessHandle`

A pointer to a variable of type HANDLE. The **ZwOpenProcess** routine writes the process handle to the variable that this parameter points to.

`DesiredAccess`

An ACCESS_MASK value that contains the access rights that the caller has requested to the process object.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the attributes to apply to the process object handle. In Windows Vista and later versions of Windows, the **ObjectName** field of this structure must be set to **NULL**. In Windows Server 2003, Windows XP, and Windows 2000, this field can, as an option, point to an object name. For more information, see the following Remarks section.

`ClientId`

A pointer to a client ID that identifies the thread whose process is to be opened. In Windows Vista and later versions of Windows, this parameter must be a non-**NULL** pointer to a valid client ID. In Windows Server 2003, Windows XP, and Windows 2000, this parameter is optional and can be set to **NULL** if the **OBJECT_ATTRIBUTES** structure that *ObjectAttributes* points to specifies an object name. For more information, see the following Remarks section.

Return Value

ZwOpenProcess returns STATUS_SUCCESS if the call is successful. Possible return values include the following error status codes.

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER_MIX	In Windows Vista and later versions of Windows, the caller either supplied an object name or failed to supply a client ID. In Windows Server 2003, Windows XP, and Windows 2000, the caller supplied both an object name and a client ID.
STATUS_INVALID_CID	The specified client ID is not valid.
STATUS_INVALID_PARAMETER	The requested access rights are not valid for a process object.
STATUS_ACCESS_DENIED	The requested access rights cannot be granted.

Remarks

In Windows Vista and later versions of Windows, the *ClientId* parameter must point to a client ID that identifies the thread whose process is to be opened. In addition, the **ObjectName** field of the structure pointed to by *ObjectAttributes* must be set to **NULL**.

In Windows Server 2003, Windows XP, and Windows 2000, the caller has the option of supplying either a client ID or an object name (but not both). If the **ObjectName** field of the structure pointed to by *ObjectAttributes* contains a non-**NULL** pointer to an object name, *ClientId* must be **NULL**.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenProcess**" instead of "**ZwOpenProcess**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

PCREATE_PROCESS_NOTIFY_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

Process-creation callback implemented by a driver to track the system-wide creation and deletion of processes against the driver's internal state.

> **Warning** The actions that you can perform in this routine are restricted for safe calls. See Best Practices.

Syntax

```
PCREATE_PROCESS_NOTIFY_ROUTINE PcreateProcessNotifyRoutine;

void PcreateProcessNotifyRoutine(
  HANDLE ParentId,
  HANDLE ProcessId,
  BOOLEAN Create
)
{...}
```

Parameters

`ParentId`

The process ID of the parent process.

`ProcessId`

The process ID of the process.

`Create`

Indicates whether the process was created (**TRUE**) or deleted (**FALSE**).

Return Value

This callback function does not return a value.

Remarks

Highest-level drivers call **PsSetCreateProcessNotifyRoutine** to register their process-creation notify routines.

A driver's process-notify routine is also called with *Create* set to **FALSE**, typically when the last thread within a process has terminated and the process address space is about to be deleted.

The operating system calls the driver's process-notify routine at PASSIVE_LEVEL inside a critical region with normal kernel APCs disabled. When a process is created, the process-notify routine runs in the context of the thread that created the new process. When a process is deleted, the process-notify routine runs in the context of the last thread to exit from the process.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL

See Also

PsSetCreateProcessNotifyRoutine

PCREATE_PROCESS_NOTIFY_ROUTINE_EX callback function

1/25/2019 • 2 minutes to read • Edit Online

A callback routine implemented by a driver to notify the caller when a process is created or exits.

Warning The actions that you can perform in this routine are restricted for safe calls. See Best Practices.

Syntax

```
PCREATE_PROCESS_NOTIFY_ROUTINE_EX PcreateProcessNotifyRoutineEx;

void PcreateProcessNotifyRoutineEx(
  PEPROCESS Process,
  HANDLE ProcessId,
  PPS_CREATE_NOTIFY_INFO CreateInfo
)
{...}
```

Parameters

`Process`

A pointer to the EPROCESS structure that represents the process. Drivers can use the **PsGetCurrentProcess** and **ObReferenceObjectByHandle** routines to obtain a pointer to the EPROCESS structure for a process.

`ProcessId`

The process ID of the process.

`CreateInfo`

A pointer to a PS_CREATE_NOTIFY_INFO structure that contains information about the new process. If this parameter is NULL, the specified process is exiting.

Return Value

This callback function does not return a value.

Remarks

Highest-level drivers call PsSetCreateProcessNotifyRoutineEx to register their implementation of *PCREATE_PROCESS_NOTIFY_ROUTINE_EX* routine. An installable file system (IFS) or highest-level system-profiling driver might register a process-creation callback routine to track which processes are created and deleted against the driver's internal state across the system.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL

See Also

PsSetCreateProcessNotifyRoutineEx

PsSetCreateProcessNotifyRoutineEx2

PCREATE_THREAD_NOTIFY_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

A callback routine implemented by a driver to notify the caller when a thread is created or deleted.

Warning The actions that you can perform in this routine are restricted for safe calls. See Best Practices.

Syntax

```
PCREATE_THREAD_NOTIFY_ROUTINE PcreateThreadNotifyRoutine;

void PcreateThreadNotifyRoutine(
  HANDLE ProcessId,
  HANDLE ThreadId,
  BOOLEAN Create
)
{...}
```

Parameters

`ProcessId`

The process ID of the process.

`ThreadId`

The thread ID of the thread.

`Create`

Indicates whether the thread was created (**TRUE**) or deleted (**FALSE**).

Return Value

This callback function does not return a value.

Remarks

Highest-level drivers can call PsSetCreateThreadNotifyRoutine or PsSetCreateThreadNotifyRoutineEx to register their thread-creation notify routine.

The driver's thread-notify routine runs at IRQL = PASSIVE_LEVEL or APC_LEVEL. When a thread is created, the thread-notify routine runs in the context of the thread that created the new thread. When a thread is deleted, the thread-notify routine runs in the context of this thread when the thread exits.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	"<=APC_LEVEL"

See Also

PsSetCreateThreadNotifyRoutine

PsSetCreateThreadNotifyRoutineEx

PGET_LOCATION_STRING callback function

1/25/2019 • 2 minutes to read • Edit Online

The *PnpGetLocationString* routine provides the device-specific part of the device's SPDRP_LOCATION_PATHS property.

Syntax

```
PGET_LOCATION_STRING PgetLocationString;

NTSTATUS PgetLocationString(
  PVOID Context,
  PZZWSTR *LocationStrings
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the PNP_LOCATION_INTERFACE structure.

`*LocationStrings`

A pointer to a sequence of null-terminated Unicode strings, that is terminated by another zero. Each string serves as a location string for the device. Drivers typically return a single string.

Return Value

The routine returns an NTSTATUS value to indicate the status of the operation.

Remarks

The PNP_LOCATION_INTERFACE structure supplies a pointer to the *PnpGetLocationString* routine.

The *PnpGetLocationString* routine provides the device-specific part of the location string for the device. The Plug and Play (PnP) manager assembles the location string for a device by querying the driver for the device, for the device's bus, and any parent buses, and concatenating the provided strings together.

The routine must return a string that is unique to the device relative to its bus. The string must be the same for the device across all versions of the operating system. Once you select a string for this purpose, you must not change it.

By convention, the location string takes the form *ServiceName(BusSpecificLocation)*. For example, PCI devices use PCI(*XXYY*), where *XX* is the device number, and *YY* is the function number.

The *PnpGetLocationString* routine calls a routine such as ExAllocatePoolWithTag to allocate the memory for the location string. The caller is responsible for calling the ExFreePool routine to free the memory pointed to by *LocationStrings* when the location string is no longer needed.

Requirements

Minimum supported client	Available starting with Windows Server 2003.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)
IRQL	"<= PASSIVE_LEVEL"

See Also

ExAllocatePoolWithTag

ExFreePool

PNP_LOCATION_INTERFACE

pHalExamineMBR callback function

1/25/2019 • 2 minutes to read • Edit Online

The **HalExamineMBR** routine reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.

Syntax

```
pHalExamineMBR Phalexaminembr;

void Phalexaminembr(
  PDEVICE_OBJECT DeviceObject,
  ULONG SectorSize,
  ULONG MBRTypeIdentifier,
  PVOID *Buffer
)
{...}
```

Parameters

`DeviceObject`

A pointer to the device object for the device being examined.

`SectorSize`

The minimum number of bytes that an I/O operation can fetch from the device being examined. If this value is less than 512, **HalExamineMBR** reads 512 bytes to ensure that it reads an entire partition table.

`MBRTypeIdentifier`

MBR partition type identifier. This parameter specifies the type of MBR that may be on the disk. For more information, see Remarks.

`*Buffer`

A pointer to a location to which **HalExamineMBR** writes a pointer to a buffer that contains data from the MBR. The layout of the buffer depends on the MBR partition type. **HalExamineMBR** allocates the storage for this buffer. The caller must deallocate this buffer as soon as possible by calling the ExFreePool routine.

HalExamineMBR sets **Buffer* = **NULL** if the MBR partition type of the disk does not match that specified by *MBRTypeIdentifier* or if there is an error.

Return Value

None

Remarks

For a list of system-defined MBR partition type identifiers, see the table in PARTITION_INFORMATION. These identifiers are defined in the Ntdddisk.h header file.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ExFreePool

PARTITION_INFORMATION

PLOAD_IMAGE_NOTIFY_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

Called by the operating system to notify the driver when a driver image or a user image (for example, a DLL or EXE) is mapped into virtual memory.

> **Warning** The actions that you can perform in this routine are restricted for safe calls. See Best Practices.

Syntax

```
PLOAD_IMAGE_NOTIFY_ROUTINE PloadImageNotifyRoutine;

void PloadImageNotifyRoutine(
  PUNICODE_STRING FullImageName,
  HANDLE ProcessId,
  PIMAGE_INFO ImageInfo
)
{...}
```

Parameters

`FullImageName`

A pointer to a buffered Unicode string that identifies the executable image file. (The *FullImageName* parameter can be **NULL** in cases in which the operating system is unable to obtain the full name of the image at process creation time.)

`ProcessId`

The process ID of the process in which the image has been mapped, but this handle is zero if the newly loaded image is a driver.

`ImageInfo`

A pointer to an IMAGE_INFO structure that contains image information. See Remarks.

Return Value

This callback function does not return a value.

Remarks

Highest-level system-profiling drivers can call PsSetLoadImageNotifyRoutine to set up their load-image notify routine.

The operating system does not call load-image notify routines when sections created with the SEC_IMAGE_NO_EXECUTE attribute are mapped to virtual memory.

In Windows 7, Windows Server 2008 R2, and earlier versions of Windows, the operating system holds an internal system lock during calls to load-image notify routines for images loaded in user process address space (user space). To avoid deadlocks, load-image notify routines must not call system routines that map, allocate, query, free, or perform other operations on user-space virtual memory.

A driver must remove any callbacks it registers before it unloads. You can remove the callback by calling the PsRemoveLoadImageNotifyRoutine routine.

When the main executable image for a newly created process is loaded, the load-image notify routine runs in the context of the new process. The operating system calls the driver's load-image notify routine at PASSIVE_LEVEL inside a critical region with normal kernel APCs always disabled and sometimes with both kernel and special APCs disabled.

```
    When the load-image notify routine is called, the input <i>FullImageName</i> points to a buffered Unicode
string that identifies the executable image file. (The <i>FullImageName</i> parameter can be <b>NULL</b> in
cases in which the operating system is unable to obtain the full name of the image at process creation time.)
The <i>ProcessId</i> handle identifies the process in which the image has been mapped, but this handle is zero
if the newly loaded image is a driver. To see the format of the buffered data at <i>ImageInfo</i>, see  <a
href="https://msdn.microsoft.com/library/windows/hardware/mt764083">IMAGE_INFO</a>. If the
<b>ExtendedInfoPresent</b> flag is set in the <b>IMAGE_INFO</b> structure, the information is part of a larger,
extended version of the image information structure, <a
href="https://msdn.microsoft.com/library/windows/hardware/mt764084">IMAGE_INFO_EX</a>.
```

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL

See Also

IMAGE_INFO

IMAGE_INFO_EX

PsSetLoadImageNotifyRoutine

PsAllocSiloContextSlot function

1/25/2019 • 2 minutes to read • Edit Online

This routine allocates a slot that can be used to insert, retrieve, and delete an object in all silos.

Syntax

```
NTSTATUS PsAllocSiloContextSlot(
  ULONG_PTR Reserved,
  ULONG     *ReturnedContextSlot
);
```

Parameters

`Reserved`

This parameter is reserved for future use and **must be set to zero**.

> **Warning** Setting *Reserved* to a non-zero value causes the system to execute an **ASSERT** on a checked build.

`ReturnedContextSlot`

A pointer to a caller-allocated variable that receives the newly allocated slot index. This parameter is required and it cannot be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INSUFFICIENT_RESOURCES	There are no more slots available in the system. This is an error code.
STATUS_SUCCESS	The operation completed successfully.

Remarks

Use of this API is uncommon because all silo monitors are assigned a storage slot when calling the PsRegisterSiloMonitor routine. That slot can be retrieved with the PsGetSiloMonitorContextSlot routine and used by a driver for its context operations.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

See Also

PsFreeSiloContextSlot

PsAttachSiloToCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

This routine places a thread temporarily into the specified *Silo*.

Syntax

```
PESILO PsAttachSiloToCurrentThread(
  PESILO Silo
);
```

Parameters

`Silo`

The silo that the thread is to impersonate. The caller must hold a reference to the silo throughout the duration of the impersonation.

Return Value

The previous silo that was attached to the current thread.

Remarks

The specified *Silo* is attached to the current thread so that it becomes the effective silo for the thread.

The thread then operates within the namespace of the attached silo until PsDetachSiloFromCurrentThread is called.

Examples

C++

```
PESILO PreviousSilo = PsAttachSiloToCurrentThread();

// Do work within the silo context

PsDetachSiloFromCurrentThread(PreviousSilo);
```

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows

Header	ntddk.h

See Also

PsDetachSiloFromCurrentThread

PsCreateSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine creates an object that will be inserted in a *Silo*.

Syntax

```
NTSTATUS PsCreateSiloContext(
  PESILO                        Silo,
  ULONG                         Size,
  POOL_TYPE                     PoolType,
  SILO_CONTEXT_CLEANUP_CALLBACK ContextCleanupCallback,
  PVOID                         *ReturnedSiloContext
);
```

Parameters

`Silo`

A pointer to a silo. This parameter is required and it cannot be **NULL**.

`Size`

The size, in bytes, of the portion of the object defined by the caller.

`PoolType`

The type of pool to allocate from. This parameter is required and must be one of the following: **NonPagedPoolNx** or **PagedPool**.

`ContextCleanupCallback`

A pointer to a SILO_CONTEXT_CLEANUP_CALLBACK callback function. The function will be called when the returned object has zero references to it. This parameter is optional and can be **NULL**.

`ReturnedSiloContext`

A pointer to a caller-allocated variable that receives the address of the newly created object.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION

STA TUS _IN SUF FICI ENT _RE SO URC ES	The routine encountered a pool allocation failure. This is an error code.
STA TUS _IN VAL ID_ PAR AM ETE R	The pool type is not valid. This is an error code.
STA TUS _SU CCE SS	The operation completed successfully.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsDereferenceSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine decrements the reference count on the object.

Syntax

```
void PsDereferenceSiloContext(
  PVOID SiloContext
);
```

Parameters

`SiloContext`

A pointer to the object created by the PsCreateSiloContext routine. This parameter is required and it cannot be **NULL**.

Return Value

This routine does not return a value.

Remarks

If the reference count reaches zero it will call the cleanup callback provided when the PsCreateSiloContext routine created the object.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
IRQL	"*IRQL_requires_max*(DISPATCH_LEVEL)"

PsDetachSiloFromCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

This routine removes a thread from a silo which was added by an attach. For more info about attaching, see the PsAttachSiloToCurrentThread routine.

> **Note** The caller is responsible for dereferencing the object after the detach has completed.

Syntax

```
void PsDetachSiloFromCurrentThread(
  PESILO PreviousSilo
);
```

Parameters

`PreviousSilo`

The value returned from the silo attach call.

Return Value

This routine does not return a value.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

See Also

PsAttachSiloToCurrentThread

PsFreeSiloContextSlot function

1/25/2019 • 2 minutes to read • Edit Online

This routine frees the specified slot and makes it available in the system. It undoes the effects of the PsAllocSiloContextSlot routine.

Syntax

```
NTSTATUS PsFreeSiloContextSlot(
  ULONG ContextSlot
);
```

Parameters

`ContextSlot`

A slot allocated by the PsAllocSiloContextSlot routine.

> **Warning** Setting this parameter to a slot that is still in use causes the system to execute bug check.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The slot is not allocated in the system. This is an error code.
STATUS_SUCCESS	The operation completed successfully.

Requirements

Minimum supported client	Windows 10, version 1607

Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

See Also

PsAllocSiloContextSlot

PsGetCurrentProcessId function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentProcessId** routine identifies the current thread's process.

Syntax

```
HANDLE PsGetCurrentProcessId(

);
```

Parameters

This function has no parameters.

Return Value

PsGetCurrentProcessId returns the process ID of the current thread's process.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoGetCurrentProcess

PsGetCurrentProcess

PsGetCurrentThread

PsGetCurrentThreadId

PsSetCreateProcessNotifyRoutine

PsSetCreateThreadNotifyRoutine

PsSetLoadImageNotifyRoutine

PsGetCurrentServerSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the effective server silo for the thread.

Syntax

```
PESILO PsGetCurrentServerSilo(

);
```

Parameters

This function has no parameters.

Return Value

A pointer to the current server silo. This pointer is valid for the current thread, but must be referenced before transferring to another thread (for example, via a workitem).

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
IRQL	"*IRQL_requires_max*(DISPATCH_LEVEL)"

PsGetCurrentSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the current silo for the calling thread. First the thread is checked to see if it has been attached to a silo. If not, then the thread is checked to see if it is in a silo.

Syntax

```
PESILO PsGetCurrentSilo(

);
```

Parameters

This function has no parameters.

Return Value

A pointer to the **ESILO** object. This pointer is valid for the current thread, but must be referenced before transferring to another thread (for example, via a workitem).

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
IRQL	"*IRQL_requires_max*(DISPATCH_LEVEL)"

PsGetCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentThread** routine identifies the current thread.

Syntax

```
PETHREAD PsGetCurrentThread(

);
```

Parameters

This function has no parameters.

Return Value

PsGetCurrentThread returns a pointer to the executive thread object that represents the currently executing thread.

Remarks

This macro

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentThread

PsCreateSystemThread

PsGetCurrentProcess

PsGetCurrentProcessId

PsIsSystemThread

PsGetCurrentThreadId function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentThreadId** routine identifies the current thread.

Syntax

```
HANDLE PsGetCurrentThreadId(

);
```

Parameters

This function has no parameters.

Return Value

The **PsGetCurrentThreadId** routine returns the thread ID of the current thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentThread

PsGetCurrentProcessId

PsGetCurrentThread

PsSetCreateThreadNotifyRoutine

PsGetCurrentThreadTeb function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentThreadTeb** routine returns the Thread Environment Block (TEB) of the current thread. The call must be made in kernel-mode.

Syntax

```
PVOID PsGetCurrentThreadTeb(

);
```

Parameters

This function has no parameters.

Return Value

A pointer to the thread environment block of the current thread. The TEB should be accessed within a try/catch exception block.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

See Also

Process and Thread Manager Routines

PsGetEffectiveServerSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine traverses the parent chain of the *Silo* until finding the effective server silo or host silo.

Syntax

```
PESILO PsGetEffectiveServerSilo(
  PESILO Silo
);
```

Parameters

`Silo`

A pointer to a silo.

Return Value

The effective server silo. If a server silo is not found, the host silo is returned. In that case, `PsIsHostSilo(ReturnValue)` would return **TRUE**.

Remarks

This routine does not fail because it always returns a silo: the server silo or the host silo.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

See Also

PsIsHostSilo

PsGetHostSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the host silo.

Syntax

```
PESILO PsGetHostSilo(

);
```

Parameters

This function has no parameters.

Return Value

The host silo.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsGetJobServerSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the effective *ServerSilo* for the job. The returned pointer is valid as long as the supplied *Job* object remains referenced.

> **Note** This returns a *ServerSilo* or a value indicating the host silo. Unlike PsGetJobSilo, it will not return an app silo, even if one is present.

Syntax

```
NTSTATUS PsGetJobServerSilo(
  PEJOB  Job,
  PESILO *ServerSilo
);
```

Parameters

`Job`

A job object.

`ServerSilo`

A pointer that receives the server silo for the job.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *Job* parameter is **NULL**.
STATUS_SUCCESS	A PESILO is returned successfully.

Remarks

STATUS_SUCCESS is returned even if a server silo is not in effect for the job. In that case, it will return the default host silo.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsGetJobSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the first job in the hierarchy that is a *Silo*. The returned pointer is valid as long as the supplied *Job* object remains referenced.

> **Note** This returns both app silos and server silos, whichever is first.

Syntax

```
NTSTATUS PsGetJobSilo(
  PEJOB  Job,
  PESILO *Silo
);
```

Parameters

`Job`

A job object.

`Silo`

A pointer that receives the silo for the job.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *Job* parameter is **NULL**.
STATUS_JOB_NO_CONTAINER	No silo is present.

STA **TUS** **_SU** **CCE** **SS**	A PESILO is returned successfully.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsGetParentSilo function

1/25/2019 • 2 minutes to read • Edit Online

Retrieves the most immediate parent silo in the hierarchy for a given job object.

Syntax

```
PESILO PsGetParentSilo(
  PEJOB Job
);
```

Parameters

`Job`

A pointer to an **EJOB** structure that represents the job object.

Return Value

A pointer to the parent silo of the job. his value may be PSP_HOST_SILO, because all silos descend from the host.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

PsGetPermanentSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine retrieves an object that was inserted in the *Silo* without incrementing the reference count.

Syntax

```
NTSTATUS PsGetPermanentSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  *ReturnedSiloContext
);
```

Parameters

`Silo`

The silo in which the object was inserted. This parameter is required and it cannot be **NULL**.

`ContextSlot`

The read-only slot that was previously allocated byPsAllocSiloContextSlot and made read-only by PsMakeSiloContextPermanent.

`ReturnedSiloContext`

A pointer to a caller-allocated variable that receives the address of the existing object. This parameter is required and it cannot be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STA TUS _NO T_F OU ND	The slot is empty. This is an error code.
STA TUS _NO T_S UPP ORT ED	The slot is not read-only and it cannot safely retrieve the object. This is an error code.

STA **TUS** **_SU** **CCE** **SS**	The operation completed successfully.

Remarks

A successful call to **PsGetPermanentSiloContext** does not increment the reference count on the object that the *ReturnedSiloContext* parameter points to. The returned object pointer is valid as long as there is a valid reference on the silo object.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsGetProcessCreateTimeQuadPart function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetProcessCreateTimeQuadPart** routine returns a LONGLONG value that represents the time at which the process was created.

Syntax

```
LONGLONG PsGetProcessCreateTimeQuadPart(
  PEPROCESS Process
);
```

Parameters

`Process`

A pointer to the EPROCESS structure that represents the process. Drivers can use the PsGetCurrentProcess and ObReferenceObjectByHandle routines to obtain a pointer to the EPROCESS structure for a process.

Return Value

PsGetProcessCreateTimeQuadPart returns the process creation time, in 100-nanosecond intervals, since January 1, 1601. The return value is the same as the value that the KeQuerySystemTime routine returns when the process was created. (Note that if the system time is changed, the value that **PsGetProcessCreateTimeQuadPart** returns is unaffected.)

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	Any level

See Also

KeQuerySystemTime

ObReferenceObjectByHandle

PsGetCurrentProcess

PsGetProcessId function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetProcessId** routine returns the process identifier (process ID) that is associated with a specified process.

Syntax

```
HANDLE PsGetProcessId(
  PEPROCESS Process
);
```

Parameters

`Process`

A pointer to a process object structure.

Return Value

PsGetProcessId returns the process ID of the process that the *Process* parameter specifies.

Remarks

The EPROCESS-typed process object structure is an opaque data structure that the operating system uses internally. To obtain a pointer to the EPROCESS structure for the current process, a driver can call PsGetCurrentProcess. To obtain a pointer to the EPROCESS structure for a different process, the driver can call ObReferenceObjectByHandle.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ObReferenceObjectByHandle

PsGetCurrentProcess

PsGetServerSiloActiveConsoleId function

1/25/2019 • 2 minutes to read • Edit Online

Gets the active console for the current server silo context for the supplied thread.

Syntax

```
ULONG PsGetServerSiloActiveConsoleId(
  PESILO Silo
);
```

Parameters

`Silo`

A pointer to the silo of the job.

Return Value

Returns the session id for the active console session.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

PsGetSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine retrieves the silo context from the specified silo and slot.

Syntax

```
NTSTATUS PsGetSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  *ReturnedSiloContext
);
```

Parameters

`Silo`

The silo where the silo context is to exist. This parameter is required and it cannot be **NULL**.

`ContextSlot`

The slot where the silo context is to exist. A slot allocated by the PsAllocSiloContextSlot routine.

`ReturnedSiloContext`

Receives a referenced pointer to the silo context. On failure, the value received will be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_NOT_FOUND	Status code if the silo context is not found.
STATUS_INVALID_PARAMETER	Status code if an invalid slot number was supplied as the *ContextSlot* parameter.

STA **TUS** **_SU** **CCE** **SS**	The operation completed successfully.

Remarks

The **PsGetSiloContext** routine retrieves an object that was inserted in the specified silo. A successful call to this routine increments the reference count on the object that the *ReturnedSiloContext* parameter points to. The object that the *ReturnedSiloContext* parameter points to, must be decremented by calling PsDereferenceSiloContext when it is no longer needed. A context slot may go empty if the silo is being terminated by either having no more processes or a specific call to **NtTerminateJobObject**. The return status in this case is **STATUS_NOT_FOUND**.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsGetSiloMonitorContextSlot function

1/25/2019 • 2 minutes to read • Edit Online

This routine returns the silo context slot that was allocated by the monitor during the registration.

Syntax

```
ULONG PsGetSiloMonitorContextSlot(
  PSILO_MONITOR Monitor
);
```

Parameters

`Monitor`

A pointer to the silo monitor.

Return Value

A valid silo context slot.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsInsertPermanentSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine inserts an object in an empty slot in a *Silo*.

Syntax

```
NTSTATUS PsInsertPermanentSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  SiloContext
);
```

Parameters

`Silo`

The silo in which the object is to be inserted. This parameter is required and it cannot be **NULL**.

`ContextSlot`

The slot in which the object is to be inserted. A slot allocated by the PsAllocSiloContextSlot routine.

`SiloContext`

The object to be inserted, created by the PsCreateSiloContext routine. The object must be created using the same silo as specified in the *Silo* parameter. This parameter is required and it cannot be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STA TUS _IN SUF FICI ENT _RE SO URC ES	There are no resources in the system to perform the insert. This is an error code.

STA TUS _NO T_S UPP ORT ED	The slot is not empty. This is an error code.
STA TUS _SU CCE SS	The operation completed successfully.

Remarks

A successful call to **PsInsertPermanentSiloContext** increments the reference count on *SiloContext*. If **PsInsertPermanentSiloContext** fails, the reference count remains unchanged. In either case, after the routine completes, the caller must call PsDereferenceSiloContext to decrement the *SiloContext* object.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsInsertSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine inserts an object in an empty slot in a *Silo*.

Syntax

```
NTSTATUS PsInsertSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  SiloContext
);
```

Parameters

`Silo`

A pointer to a silo. This parameter is required and it cannot be **NULL**.

`ContextSlot`

A slot allocated by the PsAllocSiloContextSlot routine.

`SiloContext`

A pointer to the object created by the PsCreateSiloContext routine. The object must be created using the same silo pointer as the one specified in this routine. This parameter is required and it cannot be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STA TUS _IN SUF FICI ENT _RE SO URC ES	There are no resources in the system to perform the insert. This is an error code.

STA TUS _NO T_S UPP ORT ED	The slot is not empty. This is an error code.
STA TUS _SU CCE SS	The operation completed successfully.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsIsHostSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine will check if the supplied *Silo* is the host silo.

Syntax

```
BOOLEAN PsIsHostSilo(
  PESILO Silo
);
```

Parameters

`Silo`

The silo to evaluate.

Return Value

Returns **true** if the specified silo is the host silo; otherwise, **false**.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsMakeSiloContextPermanent function

1/25/2019 • 2 minutes to read • Edit Online

This routine makes the slot in a silo instance read-only, allowing the object in the slot to be retrieved without affecting the reference count on that object.

Syntax

```
NTSTATUS PsMakeSiloContextPermanent(
  PESILO Silo,
  ULONG  ContextSlot
);
```

Parameters

`Silo`

The silo in which the slot resides. This parameter is required and it cannot be **NULL**.

`ContextSlot`

The slot to make read-only. The slot must be previously allocated by the PsAllocSiloContextSlot routine.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The slot does not contain a valid object. This is an error code.
STATUS_NOT_FOUND	The slot has not been allocated. This is an error code.

STA **TUS** **_SU** **CCE** **SS**	The operation completed successfully.

Remarks

Before calling this routine, the slot must contain a valid object. After it completes, the PsReplaceSiloContext and PsRemoveSiloContext routines will fail with **STATUS_NOT_SUPPORTED**.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsReferenceSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine increments the reference count on the object.

Syntax

```
void PsReferenceSiloContext(
  PVOID SiloContext
);
```

Parameters

`SiloContext`

A pointer to the object created by the PsCreateSiloContext routine. This parameter is required and it cannot be **NULL**.

Return Value

This routine does not return a value.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
IRQL	"*IRQL_requires_max*(DISPATCH_LEVEL)"

See Also

PsCreateSiloContext

PsRegisterSiloMonitor function

1/25/2019 • 2 minutes to read • Edit Online

This routine registers a server silo monitor that can receive notifications about server silo events.

> **Note** To start receiving notifications, call the PsStartSiloMonitor routine.

Syntax

```
NTSTATUS PsRegisterSiloMonitor(
  PSILO_MONITOR_REGISTRATION Registration,
  PSILO_MONITOR              *ReturnedMonitor
);
```

Parameters

`Registration`

Specifies the server silo monitor to be registered, of type SILO_MONITOR_REGISTRATION.

`ReturnedMonitor`

Receives a pointer to the monitor. This pointer is used to make further monitor-related calls.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The version specified in **ntddk.h** does not match **SILO_MONITOR_REGISTRATION_VERSION**, the component name is not specified, or the terminate callback is not supplied.
STATUS_PRIVILEGE_NOT_HELD	The routine is called in a silo.

STA TUS _IN SUF FICI ENT _RE SO URC ES	There is no memory to register a silo monitor or there is no available silo slot.
STA TUS _SU CCE SS	The operation completed successfully.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsRemoveCreateThreadNotifyRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **PsRemoveCreateThreadNotifyRoutine** routine removes a callback routine that was registered by the PsSetCreateThreadNotifyRoutine routine.

Syntax

```
NTSTATUS PsRemoveCreateThreadNotifyRoutine(
  PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
);
```

Parameters

`NotifyRoutine`

Pointer to the callback routine that the driver has previously registered through PsSetCreateThreadNotifyRoutine.

Return Value

PsRemoveCreateThreadNotifyRoutine returns STATUS_SUCCESS if it successfully removes the callback routine, or STATUS_PROCEDURE_NOT_FOUND if the value of *NotifyRoutine* does not match any registered callback routine.

Remarks

If the driver's callback routine is currently running, **PsRemoveCreateThreadNotifyRoutine** waits until the callback routine exits before removing it. Therefore, the callback routine itself must not call **PsRemoveCreateThreadNotifyRoutine**.

Requirements

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PsSetCreateThreadNotifyRoutine

PsRemoveLoadImageNotifyRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **PsRemoveLoadImageNotifyRoutine** routine removes a callback routine that was registered by the PsSetLoadImageNotifyRoutine routine.

Syntax

```
NTSTATUS PsRemoveLoadImageNotifyRoutine(
  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine
);
```

Parameters

`NotifyRoutine`

Pointer to the callback routine that the driver has previously registered through PsSetLoadImageNotifyRoutine.

Return Value

PsRemoveLoadImageNotifyRoutine returns STATUS_SUCCESS if it successfully removes the callback routine, or STATUS_PROCEDURE_NOT_FOUND if the value of *NotifyRoutine* does not match any registered callback routine.

Remarks

If the driver's callback routine is currently running, **PsRemoveLoadImageNotifyRoutine** waits until the callback routine exits before removing it. Therefore, the callback routine itself must not call **PsRemoveLoadImageNotifyRoutine**.

Requirements

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PsSetLoadImageNotifyRoutine

PsRemoveSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine removes an object that was inserted in the *Silo*.

Syntax

```
NTSTATUS PsRemoveSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  *RemovedSiloContext
);
```

Parameters

`Silo`

The silo from which the object is to be removed. This parameter is required and it cannot be **NULL**.

`ContextSlot`

A slot allocated by the PsAllocSiloContextSlot routine.

`RemovedSiloContext`

A pointer to a caller-allocated variable that receives the address of the removed object. This parameter is optional and can be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_NOT_FOUND	The slot is empty. This is an error code.
STATUS_UPPORTED	The slot is read-only and it cannot be modified. This is an error code.

STA **TUS** **_SU** **CCE** **SS**	The operation completed successfully.

Remarks

In a successful call to **PsRemoveSiloContext** where the *RemovedSiloContext* parameter is not **NULL** and does not point to **NULL**, the caller must decrement the object that the *RemovedSiloContext* parameter points to, by calling PsDereferenceSiloContext when it is no longer needed.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsReplaceSiloContext function

1/25/2019 • 2 minutes to read • Edit Online

This routine inserts an object in a *Silo*.

Syntax

```
NTSTATUS PsReplaceSiloContext(
  PESILO Silo,
  ULONG  ContextSlot,
  PVOID  NewSiloContext,
  PVOID  *OldSiloContext
);
```

Parameters

`Silo`

A pointer to a silo. This parameter is required and it cannot be **NULL**.

`ContextSlot`

A slot allocated by the PsAllocSiloContextSlot routine.

`NewSiloContext`

A pointer to the object created by the PsCreateSiloContext routine. The object must be created using the same silo pointer as the one specified in this routine. This parameter is required and it cannot be **NULL**.

`OldSiloContext`

A pointer to a caller-allocated variable that receives the address of the existing object. This parameter is optional and can be **NULL**. The address that the parameter receives can be **NULL**.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STA TUS _IN SUF FICI ENT _RE SO URC ES	There are no resources in the system to perform the insert. This is an error code.

STA TUS _NO T_S UPP ORT ED	The slot is read-only and it cannot be modified. This is an error code.
STA TUS _SU CCE SS	The operation completed successfully.

Remarks

A successful call to **PsReplaceSiloContext** increments the reference count on *NewSiloContext*. If **PsReplaceSiloContext** fails, the reference count remains unchanged. In either case, the caller of **PsReplaceSiloContext** must call PsDereferenceSiloContext to decrement the **PsReplaceSiloContext** object. If **PsReplaceSiloContext** fails and if the *OldSiloContext* parameter is not **NULL** and does not point to **NULL** then **NULL** is a referenced pointer. After the routine completes, the caller must call **PsDereferenceSiloContext** to decrement the object that the **NULL** parameter points to.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsSetCreateProcessNotifyRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetCreateProcessNotifyRoutine** routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

Syntax

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
  PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
  BOOLEAN                        Remove
);
```

Parameters

`NotifyRoutine`

Specifies the entry point of a caller-supplied process-creation callback routine. See PCREATE_PROCESS_NOTIFY_ROUTINE.

`Remove`

Indicates whether the routine specified by *NotifyRoutine* should be added to or removed from the system's list of notification routines. If **FALSE**, the specified routine is added to the list. If **TRUE**, the specified routine is removed from the list.

Return Value

PsSetCreateProcessNotifyRoutine can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The given *NotifyRoutine* is now registered with the system.
STATUS_INVALID_PARAMETER	The given *NotifyRoutine* has already been registered, so this call is a redundant call, or the system has reached its limit for registering process-creation callbacks.

Remarks

Highest-level drivers can call **PsSetCreateProcessNotifyRoutine** to set up their process-creation notify routines implemented as PCREATE_PROCESS_NOTIFY_ROUTINE.

An IFS or highest-level system-profiling driver might register a process-creation callback to track the system-wide creation and deletion of processes against the driver's internal state. For Windows Vista and later versions of Windows, the system can register up to 64 process-creation callback routines.

A driver must remove any callbacks that it registers before it unloads. You can remove the callback by calling **PsSetCreateProcessNotify** with *Remove* = **TRUE**.

After a driver-supplied routine is registered, it is called with *Create* set to **TRUE** just after the initial thread is created within the newly created process designated by the input *ProcessId* handle. The input *ParentId* handle identifies the parent process of the newly-created process (this is the parent used for priority, affinity, quota, token, and handle inheritance, among others).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlPsPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PCREATE_PROCESS_NOTIFY_ROUTINE

PsGetCurrentProcessId

PsSetCreateProcessNotifyRoutineEx

PsSetCreateThreadNotifyRoutine

PsSetLoadImageNotifyRoutine

PsSetCreateProcessNotifyRoutineEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetCreateProcessNotifyRoutineEx** routine registers or removes a callback routine that notifies the caller when a process is created or exits.

Syntax

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx(
  PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,
  BOOLEAN                           Remove
);
```

Parameters

`NotifyRoutine`

A pointer to the PCREATE_PROCESS_NOTIFY_ROUTINE_EX routine to register or remove. The operating system calls this routine whenever a new process is created.

`Remove`

A Boolean value that specifies whether **PsSetCreateProcessNotifyRoutineEx** will add or remove a specified routine from the list of callback routines. If this parameter is **TRUE**, the specified routine is removed from the list of callback routines. If this parameter is **FALSE**, the specified routine is added to the list of callback routines. If *Remove* is **TRUE**, the system also waits for all in-flight callback routines to complete before returning.

Return Value

PsSetCreateProcessNotifyRoutineEx returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The specified routine is now registered with the operating system. The operating system calls this routine whenever a new process is created.
STATUS_INVALID_PARAMETER	The specified routine was already registered, or the operating system has reached its limit for registering process-creation callback routines.

STA TUS _AC CES S_D ENI ED	The image that contains the callback routine pointer did not have IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY set in its image header.

Remarks

Highest-level drivers can call **PsSetCreateProcessNotifyRoutineEx** to register a PCREATE_PROCESS_NOTIFY_ROUTINE_EX routine. An installable file system (IFS) or highest-level system-profiling driver might register a process-creation callback routine to track which processes are created and deleted against the driver's internal state across the system.

A driver must remove any callback routines that it registers before it unloads. You can remove the callback routine by calling **PsSetCreateProcessNotifyRoutineEx** with *Remove* set to **TRUE**.

The operating system calls the driver's process-notify routine at PASSIVE_LEVEL inside a critical region with normal kernel APCs disabled. When a process is created, the process-notify routine runs in the context of the thread that created the new process. When a process is deleted, the process-notify routine runs in the context of the last thread to exit from the process.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1 and Windows Server 2008.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PCREATE_PROCESS_NOTIFY_ROUTINE_EX

PS_CREATE_NOTIFY_INFO

PsSetCreateProcessNotifyRoutine

PsSetCreateProcessNotifyRoutineEx2 function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetCreateProcessNotifyRoutineEx2** routine registers or removes a callback routine that notifies the caller when a process is created or deleted.

Syntax

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx2(
  PSCREATEPROCESSNOTIFYTYPE NotifyType,
  PVOID                     NotifyInformation,
  BOOLEAN                   Remove
);
```

Parameters

`NotifyType`

A PSCREATEPROCESSNOTIFYTYPE-type value that indicates the type of process notification.

`NotifyInformation`

The address of the notification information for the specified type of process notification. If *NotifyType* is **PsCreateProcessNotifySubsystems**, *NotifyInformation* is a PCREATE_PROCESS_NOTIFY_ROUTINE_EX that specifies the entry point of the caller-supplied process-creation callback.

`Remove`

A Boolean value that specifies whether **PsSetCreateProcessNotifyRoutineEx2** will add or remove a specified routine from the list of callback routines. If this parameter is **TRUE**, the specified routine is removed from the list of callback routines. If this parameter is **FALSE**, the specified routine is added to the list of callback routines. If *Remove* is **TRUE**, the system also waits for all in-flight callback routines to complete before returning.

Return Value

PsSetCreateProcessNotifyRoutineEx2 returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The specified routine is now registered with the operating system. The operating system calls this routine whenever a new process is created.

STATUS_INVALID_PARAMETER	The specified routine was already registered, or the operating system has reached its limit for registering process-creation callback routines. *NotifyType* is not **PsCreateProcessNotifySubsystems**.
STATUS_ACCESS_DISENIED	The image that contains the callback routine pointer did not have IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY set in its image header.

Remarks

Drivers can call **PsSetCreateProcessNotifyRoutineEx2** to register their process-creation notify routines.

After a driver-supplied routine is registered, it is called with the unique ID (indicated by *ProcessId*) of the created or deleted process. The *ParentId* identifies the parent process of the new process (this is the parent used for priority, affinity, quota, token, and handle inheritance, among others) if it was created with the inherit handles option. If it was created without the inherit handle options, then the parent process ID is NULL.

If the *Create* value is TRUE, the subsystem process was created; FALSE indicates the process was deleted.

When the process is created, the callback function is invoked just after the first thread in the process has been created. Conversely, for deletion, the function is invoked after the last thread in the process has terminated and the address space is about to be deleted. It is possible that the callback is only invoked for deletion without getting a creation call in cases where the process was created and deleted without a thread ever being created.

A driver must remove any callback function that it registers before it unloads. You can remove the callback by calling **PsSetCreateProcessNotifyRoutineEx2** with *Remove* = **TRUE**.

Requirements

Minimum supported client	Windows 10, version 1703
Minimum supported server	Windows Server 2016
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PCREATE_PROCESS_NOTIFY_ROUTINE_EX

PsSetCreateProcessNotifyRoutine

PsSetCreateProcessNotifyRoutineEx

PsSetCreateThreadNotifyRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetCreateThreadNotifyRoutine** routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.

Syntax

```
NTSTATUS PsSetCreateThreadNotifyRoutine(
  PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
);
```

Parameters

`NotifyRoutine`

A pointer to the driver's implementation of PCREATE_THREAD_NOTIFY_ROUTINE routine.

Return Value

PsSetCreateThreadNotifyRoutine either returns STATUS_SUCCESS or it returns STATUS_INSUFFICIENT_RESOURCES if it failed the callback registration.

Remarks

Highest-level drivers can call **PsSetCreateThreadNotifyRoutine** to set up their thread-creation notify routines, declared as follows:

```
VOID
(*PCREATE_THREAD_NOTIFY_ROUTINE) (
    IN HANDLE  ProcessId,
    IN HANDLE  ThreadId,
    IN BOOLEAN  Create
    );
```

For example, an IFS or highest-level system-profiling driver might register such a thread-creation callback to track the system-wide creation and deletion of threads against the driver's internal state.

A driver must remove any callbacks it registers before it unloads. You can remove the callback by calling the PsRemoveCreateThreadNotifyRoutine routine.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlPsPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PsGetCurrentProcessId

PsGetCurrentThreadId

PsIsSystemThread

PsRemoveCreateThreadNotifyRoutine

PsSetCreateProcessNotifyRoutine

PsSetLoadImageNotifyRoutine

PsSetCreateThreadNotifyRoutineEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetCreateThreadNotifyRoutineEx** routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.

Syntax

```
NTSTATUS PsSetCreateThreadNotifyRoutineEx(
  PSCREATETHREADNOTIFYTYPE NotifyType,
  PVOID                   NotifyInformation
);
```

Parameters

`NotifyType`

A PSCREATETHREADNOTIFYTYPE value that indicates the type of thread notification.

`NotifyInformation`

Provides the address of the notification information for the specified type of thread notification.

Return Value

PsSetCreateThreadNotifyRoutineEx either returns STATUS_SUCCESS or it returns STATUS_INSUFFICIENT_RESOURCES if it failed the callback registration.

Remarks

Highest-level drivers can call **PsSetCreateThreadNotifyRoutineEx** to set up their thread-creation notify routines.

For example, an IFS or highest-level system-profiling driver might register such a thread-creation callback to track the system-wide creation and deletion of threads against the driver's internal state.

If *NotifyType* is **PsCreateThreadNotifyNonSystem**, the **PsSetCreateThreadNotifyRoutineEx** routine differs from PsSetCreateThreadNotifyRoutine in the context in which the callback is executed. With **PsSetCreateThreadNotifyRoutine**, the callback is executed on the creator thread. With **PsSetCreateThreadNotifyRoutineEx**, the callback is executed on the newly created thread.

A driver must remove any callback function it registers before it unloads. You can remove the callback by calling the PsRemoveCreateThreadNotifyRoutine routine.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal

Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

PsGetCurrentProcessId

PsGetCurrentThreadId

PsIsSystemThread

PsRemoveCreateThreadNotifyRoutine

PsSetCreateProcessNotifyRoutine

PsSetCreateThreadNotifyRoutine

PsSetLoadImageNotifyRoutine

PsSetLoadImageNotifyRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetLoadImageNotifyRoutine** routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).

Syntax

```
NTSTATUS PsSetLoadImageNotifyRoutine(
  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine
);
```

Parameters

`NotifyRoutine`

A pointer to the caller-implemented PLOAD_IMAGE_NOTIFY_ROUTINE callback routine for load-image notifications.

Return Value

PsSetLoadImageNotifyRoutine either returns STATUS_SUCCESS or it returns STATUS_INSUFFICIENT_RESOURCES if it failed the callback registration.

Remarks

Highest-level system-profiling drivers can call **PsSetLoadImageNotifyRoutine** to set up their load-image notify routines (see PLOAD_IMAGE_NOTIFY_ROUTINE).

The maximum number of drivers that can be simultaneously registered to receive load-image notifications is eight. If the maximum number of load-image notify routines is already registered when a driver calls **PsSetLoadImageNotifyRoutine** to try to register an additional notify routine, **PsSetLoadImageNotifyRoutine** fails and returns STATUS_INSUFFICIENT_RESOURCES.

Notes

- An update for Windows 8.1 increases the maximum number of drivers registered to receive load-image notifications from eight to 64. This update is installed as part of a cumulative update that is available through Windows Update starting on April 8, 2014. In addition, this cumulative update is available at https://support.microsoft.com/kb/2919355.
- Users of Windows 7 with Service Pack 1 (SP1) can install a hotfix to increase the maximum number of drivers registered to receive load-image notifications from eight to 64. This hotfix is available at https://support.microsoft.com/kb/2922790.

A driver must remove any callbacks it registers before it unloads. You can remove the callback by calling the PsRemoveLoadImageNotifyRoutine routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlPsPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PLOAD_IMAGE_NOTIFY_ROUTINE

PsGetCurrentProcessId

PsRemoveLoadImageNotifyRoutine

PsSetCreateProcessNotifyRoutine

PsSetCreateThreadNotifyRoutine

PsSetLoadImageNotifyRoutineEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PsSetLoadImageNotifyRoutineEx** routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).

Syntax

```
NTSTATUS PsSetLoadImageNotifyRoutineEx(
  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine,
  ULONG_PTR                  Flags
);
```

Parameters

`NotifyRoutine`

A pointer to the caller-implemented PLOAD_IMAGE_NOTIFY_ROUTINE callback routine for load-image notifications.

`Flags`

Supplies a bitmask of flags that control the callback function. Here are the possible values:

- PS_IMAGE_NOTIFY_CONFLICTING_ARCHITECTURE indicates that the callback routine should be invoked for all potentially executable images, including images that have a different architecture from the native architecture of the operating system.

Return Value

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The callback was successfully registered.
STATUS_INVALID_PARAMETER_2	Invalid flag was supplied in *Flags*.

STA TUS _IN SUF FICI ENT _RE SO URC ES	The routine failed allocate a callback block due to lack of resources.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

PsStartSiloMonitor function

1/25/2019 • 2 minutes to read • Edit Online

This routine tries to start the server silo monitor.

Syntax

```
NTSTATUS PsStartSiloMonitor(
  PSILO_MONITOR Monitor
);
```

Parameters

`Monitor`

A pointer to the silo monitor.

Return Value

The following NT status codes are returned.

RETURN CODE	DESCRIPTION
STATUS_ABORTED	The monitor was not able to start.
STATUS_NOT_SUPPORTED	The monitor does not allow existing silos and one was found in the system.
STATUS_SUCCESS	The operation completed successfully.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsTerminateServerSilo function

1/25/2019 • 2 minutes to read • Edit Online

This routine terminates the specified silo.

Syntax

```
void PsTerminateServerSilo(
  PESILO   ServerSilo,
  NTSTATUS ExitStatus
);
```

Parameters

`ServerSilo`

A pointer to the silo being terminated.

`ExitStatus`

The exit status for the silo.

Return Value

This routine does not return a value.

Remarks

This routine can be called within or from outside a silo context. Note that this is different from a BugCheck; this routine will return to the caller.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

PsUnregisterSiloMonitor function

1/25/2019 • 2 minutes to read • Edit Online

This routine unregisters a server silo monitor.

Syntax

```
void PsUnregisterSiloMonitor(
  PSILO_MONITOR Monitor
);
```

Parameters

`Monitor`

The server silo monitor to unregister.

Return Value

This routine does not return a value.

Remarks

The monitor will not receive further notifications after this routine completes.

If the monitor allocated a silo context slot, this routine will not complete until all silo contexts have been removed from slot.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

RTL_RUN_ONCE_INIT_FN callback function

1/25/2019 • 2 minutes to read • Edit Online

The *RunOnceInitialization* routine performs a one-time initialization operation.

Syntax

```
RTL_RUN_ONCE_INIT_FN RtlRunOnceInitFn;

ULONG RtlRunOnceInitFn(
  PRTL_RUN_ONCE RunOnce,
  PVOID Context,
  PVOID *Parameter
)
{...}
```

Parameters

`RunOnce`

A pointer to the RTL_RUN_ONCE one-time initialization structure that the driver previously passed as a parameter to the RtlRunOnceExecuteOnce routine that called this *RunOnceInitialization* routine.

`Context`

A pointer to a PVOID variable to which the routine writes the initialization data.

`Parameter`

The *Parameter* value that the driver passed to the **RtlRunOnceExecuteOnce** routine that called this *RunOnceInitialization* routine.

Return Value

The *RunOnceInitialization* routine returns a nonzero value to indicate success, and returns zero to indicate failure.

Remarks

A driver-implemented *RunOnceInitialization* routine performs driver-specific initialization and then writes the initialization data to the memory location that the *Context* parameter points to. Drivers use the **RtlRunOnceExecuteOnce** routine to make a one-time call to a *RunOnceInitialization* routine. The operating system guarantees that the *RunOnceInitialization* routine is not called twice for the same value of the *RunOnce* parameter.

A small number of low-order bits in the location pointed to by the *Context* parameter are reserved for use by the operating system. Your driver's *RunOnceInitialization* routine should set these reserved bits to zero in the output value that it writes to **Context*. The RTL_RUN_ONCE_CTX_RESERVED_BITS constant defined in Ntddk.h specifies the number of low-order bits that are reserved. Currently, RTL_RUN_ONCE_CTX_RESERVED_BITS is defined to be two, which means that the driver must set the two least significant bits of the PVOID value pointed to by *Context* to zero.

Examples

To define a *RunOnceInitialization* callback routine, you must first provide a function declaration that identifies the

type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *RunOnceInitialization* callback routine that is named `MyRunOnceInitialization`, use the RTL_RUN_ONCE_INIT_FN type as shown in this code example:

```
RTL_RUN_ONCE_INIT_FN MyRunOnceInitialization;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
ULONG
  MyRunOnceInitialization(
    PRTL_RUN_ONCE  RunOnce,
    PVOID  Parameter,
    PVOID  *Context
    )
  {
      // Function body
  }
```

The RTL_RUN_ONCE_INIT_FN function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the RTL_RUN_ONCE_INIT_FN function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Drivers can implement the RunOnceInitialization routine only on Windows Vista and later versions of Windows.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= APC_LEVEL.

See Also

RTL_RUN_ONCE

RtlRunOnceBeginInitialize

RtlRunOnceComplete

RtlRunOnceExecuteOnce

RtlRunOnceInitialize

RtlCharToInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCharToInteger** routine converts a single-byte character string to an integer value in the specified base.

Syntax

```
NTSYSAPI NTSTATUS RtlCharToInteger(
  PCSZ   String,
  ULONG  Base,
  PULONG Value
);
```

Parameters

`String`

Pointer to a null-terminated, single-byte character string.

`Base`

Specifies decimal, binary, octal, or hexadecimal base. If this parameter is not given, the routine will look for 0x, 0o, and 0b prefixes in the input string to determine if the base should be decimal (default), binary, octal, or hexadecimal.

`Value`

Pointer to a location to which the converted value is returned.

Return Value

RtlCharToInteger returns STATUS_SUCCESS if the given character string is converted. Otherwise, it can return STATUS_INVALID_PARAMETER.

Remarks

RtlCharToInteger converts ANSI alphanumeric characters.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlInitString

RtlIntegerToUnicodeString

RtlCompareString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCompareString** routine compares two counted strings.

Syntax

```
NTSYSAPI LONG RtlCompareString(
  const STRING *String1,
  const STRING *String2,
  BOOLEAN      CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first string.

`String2`

Pointer to the second string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlCompareString returns a signed value that gives the results of the comparison:

RETURN CODE	DESCRIPTION
Zero	*String1* equals *String2*.
< Zero	*String1* is less than *String2*.
> Zero	*String1* is greater than *String2*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlCompareUnicodeString

RtlEqualString

RtlConvertLongToLuid function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlConvertLongToLuid** routine converts a long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.

Syntax

```
LUID NTAPI_INLINE RtlConvertLongToLuid(
  LONG Long
);
```

Parameters

`Long`

Specifies the long integer to convert.

Return Value

RtlConvertLongToLuid returns the converted LUID.

Remarks

RtlConvertLongToLuid is used to convert a system-defined privilege value to the locally unique identifier (LUID) used by the system to represent that privilege. Drivers typically pass a LUID to **SeSinglePrivilegeCheck**, which is usually called by network transport drivers.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

LUID

RtlConvertULongToLuid

RtlEqualLuid

SeSinglePrivilegeCheck

RtlConvertUlongToLuid function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlConvertUlongToLuid** routine converts an unsigned long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.

Syntax

```
LUID NTAPI_INLINE RtlConvertUlongToLuid(
  ULONG Ulong
);
```

Parameters

`Ulong`

Specifies the unsigned long integer to convert.

Return Value

RtlConvertUlongToLuid returns the converted LUID.

Remarks

RtlConvertUlongToLuid is used to convert a system-defined privilege value, passed as a ULONG, to a locally unique identifier (LUID) used by the system to represent that privilege. Drivers typically pass a LUID to **SeSinglePrivilegeCheck**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	ntddk.h (include Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

LUID

RtlConvertLongToLuid

RtlEqualLuid

SeSinglePrivilegeCheck

RtlCopyString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCopyString** routine copies a source string to a destination string.

Syntax

```
NTSYSAPI VOID RtlCopyString(
  PSTRING      DestinationString,
  const STRING *SourceString
);
```

Parameters

`DestinationString`

A pointer to the destination string buffer.

`SourceString`

A pointer to the source string buffer.

Return Value

None

Remarks

The*DestinationString***Length** is set to zero if no source string is supplied. The **MaximumLength** and **Buffer** members of the *DestinationString* are not modified by this routine.

The number of bytes copied from the *SourceString* is either the length of *SourceString* or the maximum length of *DestinationString*, whichever is smaller.

The *DestinationString* and *SourceString* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level (See Remarks section)

See Also

RtlCopyUnicodeString

RtlEqualString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlEqualString** routine compares two counted strings to determine whether they are equal.

Syntax

```
NTSYSAPI BOOLEAN RtlEqualString(
  const STRING *String1,
  const STRING *String2,
  BOOLEAN      CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first string.

`String2`

Pointer to the second string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlEqualString returns **TRUE** if the two strings are equal, otherwise it returns **FALSE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlEqualUnicodeString

RtlExtendCorrelationVector function

1/25/2019 • 2 minutes to read • Edit Online

This routine extends the supplied correlation vector. For a correlation vector of the form X.i, the extended value is X.i.0.

Syntax

```
NTSYSAPI NTSTATUS RtlExtendCorrelationVector(
  PCORRELATION_VECTOR CorrelationVector
);
```

Parameters

`CorrelationVector`

A pointer to a CORRELATION_VECTOR structure that represents the correlation vector to be extended.

Return Value

Returns an NTSTATUS value that indicates the success of failure of the operation.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The correlation vector was successfully incremented.
STATUS_INVALID_PARAMETER	Extending the correlation vector resulted in a buffer overflow because as the extended value is no longer a valid correlation vector.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

RtlGetEnabledExtendedFeatures function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlGetEnabledExtendedFeatures** routine returns a mask of extended processor features that are enabled by the system.

Syntax

```
NTSYSAPI ULONG64 RtlGetEnabledExtendedFeatures(
  ULONG64 FeatureMask
);
```

Parameters

`FeatureMask`

A 64-bit feature mask. This parameter indicates a set of extended processor features for which the caller requests information about whether the features are enabled. If a mask bit is one, the caller requests information about the feature that corresponds to this mask bit. The caller sets all other mask bits to zero. Set this parameter to (ULONG64)(-1) to get the mask of all enabled features. To determine whether a particular set of features is enabled, set this parameter to the bitwise OR of one or more of the following **XSTATE_MASK_*XXX*** flag bits:

VALUE	MEANING
XSTATE_MASK_LEGACY_FLOATING_POINT	The floating-point extension (x87/MMX).
XSTATE_MASK_LEGACY_SSE	The streaming SIMD extension (SSE).

XST ATE _M ASK _LE GA CY	Both the x87/MMX and SSE extensions.
XST ATE _M ASK _GS SE	The Intel Sandy Bridge (formerly Gesher) SSE extension.

Return Value

RtlGetEnabledExtendedFeatures returns a 64-bitmask of enabled extended processor features. The routine calculates this mask as the intersection (bitwise AND) between all enabled features and the value of the *FeatureMask* parameter. For more information about the features that are indicated by this return value, see XSTATE_MASK_XXX.

RtlGetEnabledExtendedFeatures returns 0 if the XSAVE and XRSTOR instructions are disabled by the system. This happens even if the system supports XSTATE_MASK_LEGACY_FLOATING_POINT and XSTATE_MASK_LEGACY_SSE features. Use **ExIsProcessorFeaturePresent** or CPUID to determine the availability of x87, MMX, and SSE processor features on x86 or x64 systems.

Remarks

Another routine, ExIsProcessorFeaturePresent, indicates whether an extended processor feature is available. Unlike **RtlGetEnabledExtendedFeatures**, however, it does not indicate whether the operating system has enabled the feature for use by kernel-mode drivers.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntdll.lib (user mode); NtosKrnl.lib (kernel mode); OneCoreUAP.lib on Windows 10
DLL	NtDll.dll (user mode); NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

See Also

ExIsProcessorFeaturePresent

RtlIncrementCorrelationVector function

1/25/2019 • 2 minutes to read • Edit Online

Increments the specified correlation vector. For a correlation vector of the form X.i, the incremented value is be X.(i+1).

Syntax

```
NTSYSAPI NTSTATUS RtlIncrementCorrelationVector(
  PCORRELATION_VECTOR CorrelationVector
);
```

Parameters

`CorrelationVector`

A pointer to a CORRELATION_VECTOR structure that represents the correlation vector to be incremented.

Return Value

Returns an NTSTATUS value that indicates the success of failure of the operation.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The correlation vector was successfully incremented.
STATUS_BUFFER_OVERFLOW	Incrementing the correlation vector resulted in a buffer overflow because as the incremented value is no longer a valid correlation vector.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows

Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

RtlInitializeCorrelationVector function

1/25/2019 • 2 minutes to read • Edit Online

Initializes the specified correlation vector with the supplied GUID.

Syntax

```
NTSYSAPI NTSTATUS RtlInitializeCorrelationVector(
  PCORRELATION_VECTOR CorrelationVector,
  int                 Version,
  const GUID          *Guid
);
```

Parameters

`CorrelationVector`

A pointer to a CORRELATION_VECTOR structure that represents the correlation vector to be initialized.

`Version`

The version of the correlation vector. Possible values are:

- RTL_CORRELATION_VECTOR_VERSION_1
- RTL_CORRELATION_VECTOR_VERSION_2
- RTL_CORRELATION_VECTOR_VERSION_CURRENT

`Guid`

The GUID to initialize the correlation vector. The first 22 bytes of the correlation vector are a base64 representation of the GUID. This value must not be NULL.

Return Value

Returns an NTSTATUS value that indicates the success of failure of the operation.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The correlation vector was successfully initialized.

STA TUS _IN VAL ID_ PAR AM ETE R	The supplied GUID is null.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

See Also

CORRELATION_VECTOR

RtlIsStateSeparationEnabled function

1/25/2019 • 2 minutes to read • Edit Online

Checks if the SKU for the current context supports multiple sessions.

Syntax

```
NTSYSAPI BOOLEAN RtlIsStateSeparationEnabled(

);
```

Parameters

This function has no parameters.

Return Value

TRUE indicates state separation enabled;

```
FALSE otherwise.
```

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

RtlMapGenericMask function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlMapGenericMask** routine determines the nongeneric access rights specified by an ACCESS_MASK.

Syntax

```
NTSYSAPI VOID RtlMapGenericMask(
  PACCESS_MASK     AccessMask,
  PGENERIC_MAPPING GenericMapping
);
```

Parameters

`AccessMask`

Pointer to an ACCESS_MASK variable. The routine updates this to contain only those access rights that are not also granted by any GENERIC_*XXX* access rights originally specified the variable. The routine also clears any GENERIC_*XXX* bits that are set.

`GenericMapping`

Pointer to a GENERIC_MAPPING structure that describes the specific access rights that correspond to each generic access right.

Return Value

None

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

GENERIC_MAPPING

RtlPrefixUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlPrefixUnicodeString** routine compares two Unicode strings to determine whether one string is a prefix of the other.

Syntax

```
NTSYSAPI BOOLEAN RtlPrefixUnicodeString(
  PCUNICODE_STRING String1,
  PCUNICODE_STRING String2,
  BOOLEAN          CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first string, which might be a prefix of the buffered Unicode string at *String2*.

`String2`

Pointer to the second string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlPrefixUnicodeString returns **TRUE** if *String1* is a prefix of *String2*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCompareUnicodeString

RtlQueryRegistryValueWithFallback function

1/25/2019 • 2 minutes to read • Edit Online

Retrieves a value entry for a registry key by using, a primary handle; if not found, uses the fallback handle.

Syntax

```
NTSYSAPI NTSTATUS RtlQueryRegistryValueWithFallback(
  HANDLE          PrimaryHandle,
  HANDLE          FallbackHandle,
  PUNICODE_STRING ValueName,
  ULONG           ValueLength,
  PULONG          ValueType,
  PVOID           ValueData,
  PULONG          ResultLength
);
```

Parameters

`PrimaryHandle`

[in] The handle to the key that is first used to query the value entries. This handle is created by a successful call to **ZwCreateKey** or **ZwOpenKey**.

`FallbackHandle`

[in] The handle to use for querying the key if value is not found using the handle specified by *PrimaryHandle*.

`ValueName`

[in] A pointer to string that contains the name of the value for which to retrieve data.

`ValueLength`

[in] Specifies the size, in bytes, of the *ValueName* string.

`ValueType`

[Out] A pointer to a ULONG variable that receives the the type of the value being queried.

`ValueData`

[out] A pointer to a buffer that receives the value data.

`ResultLength`

[out] A pointer to a variable that receives the value data size, in bytes. If the return value is STATUS_SUCCESS, indicates the number of bytes written to the *ValueData* buffer. If the call returns STATUS_BUFFER_OVERFLOW, indicates the number of bytes required. Callers can use that value to determine the size of *ValueData* buffer required to hold the data information.

Return Value

Returns STATUS_SUCCESS on success, or the appropriate error code on failure.

Requirements

Minimum supported client	Windows 10, version 1809
Header	ntddk.h (include wdm.h)
Library	NtosKrnl.lib
IRQL	PASSIVE_LEVEL

See Also

Using the Registry in a Driver

RtlRunOnceBeginInitialize function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlRunOnceBeginInitialize** routine begins a one-time initialization.

Syntax

```
NTSYSAPI NTSTATUS RtlRunOnceBeginInitialize(
  PRTL_RUN_ONCE RunOnce,
  ULONG         Flags,
  PVOID         *Context
);
```

Parameters

`RunOnce`

Pointer to the RTL_RUN_ONCE one-time initialization structure.

`Flags`

Drivers can optionally specify one or more of the following flags:

RTL_RUN_ONCE_ASYNC

Perform initialization asynchronously. The driver can perform multiple completion attempts in parallel. If this flag is used, subsequent calls to this routine will fail unless this flag is also specified.

RTL_RUN_ONCE_CHECK_ONLY

Do not begin initialization, but check to determine if initialization has already occurred. If **RtlRunOnceBeginInitialize** returns STATUS_SUCCESS, the initialization succeeded, and **Context* contains the initialized data.

`Context`

Specifies a pointer to a PVOID variable that receives the initialized data. The value of **Context* is valid only when the routine returns STATUS_SUCCESS.

Return Value

RtlRunOnceBeginInitialize returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The one-time initialization has already completed. The initialized data is stored in the memory location that *Context* points to.

STA **TUS** **_PE** **NDI** **NG**	The caller has successfully begun one-time initialization. The caller now performs the driver-specific initialization steps and then calls RtlRunOnceComplete to complete the initialization.

Remarks

Drivers can alternatively perform one-time initialization by calling RtlRunOnceExecuteOnce and supplying a RunOnceInitialization routine.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

RTL_RUN_ONCE

RtlRunOnceComplete

RtlRunOnceExecuteOnce

RtlRunOnceInitialize

RunOnceInitialization

RtlRunOnceComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlRunOnceComplete** routine completes the one-time initialization began by RtlRunOnceBeginInitialize.

Syntax

```
NTSYSAPI NTSTATUS RtlRunOnceComplete(
  PRTL_RUN_ONCE RunOnce,
  ULONG         Flags,
  PVOID         Context
);
```

Parameters

`RunOnce`

Pointer to the RTL_RUN_ONCE one-time initialization structure.

`Flags`

Drivers can optionally specify one or more of the following flags:

RTL_RUN_ONCE_ASYNC

Operate in asynchronous mode. This mode enables multiple completion attempts to execute in parallel. If this flag is used, subsequent calls to the **RtlRunOnceComplete** routine will fail unless this flag is also specified.

RTL_RUN_ONCE_INIT_FAILED

The initialization attempt failed.

`Context`

Specifies the initialized data.

Return Value

RtlRunOnceComplete returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation completed successfully.

STA TUS _UN SUC CES SFU L	The operation could not be completed. If the caller specified RTL_RUN_ONCE_ASYNC in the *Flags* parameter, this value can indicate that another thread completed the initialization.

Remarks

If **RtlRunOnceComplete** returns STATUS_SUCCESS, any subsequent call to RtlRunOnceBeginInitialize for the same RTL_RUN_ONCE structure supplies *Context* as the initialized data.

If the caller specified RTL_RUN_ONCE_ASYNC in the *Flags* parameter and **RtlRunOnceComplete** returns any value other than STATUS_SUCCESS, the caller must clean up any initialization that it attempted.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

RTL_RUN_ONCE

RtlRunOnceBeginInitialize

RtlRunOnceExecuteOnce

RtlRunOnceInitialize

RunOnceInitialization

RtlRunOnceExecuteOnce function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlRunOnceExecuteOnce** performs a one-time initialization.

Syntax

```
NTSYSAPI NTSTATUS RtlRunOnceExecuteOnce(
  PRTL_RUN_ONCE         RunOnce,
  PRTL_RUN_ONCE_INIT_FN InitFn,
  PVOID                 Context,
  PVOID                 *Parameter
);
```

Parameters

`RunOnce`

A pointer to the RTL_RUN_ONCE one-time initialization structure.

`InitFn`

A pointer to a RunOnceInitialization routine.

`Context`

A pointer to a PVOID variable that receives the initialized data.

`Parameter`

The value to pass as the *Parameter* parameter to the *RunOnceInitialization* routine.

Return Value

RtlRunOnceExecuteOnce returns STATUS_SUCCESS if the operation succeeds, or the appropriate NTSTATUS error code if the operation fails.

Remarks

For the first call to **RtlRunOnceExecuteOnce** for a particular RTL_RUN_ONCE structure, **RtlRunOnceExecuteOnce** calls the RunOnceInitialization routine to initialize the data. Every subsequent call to **RtlRunOnceExecuteOnce** for that structure supplies the same initialized data. The *RunOnceInitialization* routine will not be called twice for the same **RTL_RUN_ONCE** structure.

RtlRunOnceExecuteOnce runs with normal kernel APCs disabled. The routine should not be called within a special kernel APC unless all calls occur at APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows Vista.

Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section.)"

See Also

RTL_RUN_ONCE

RtlRunOnceBeginInitialize

RtlRunOnceComplete

RtlRunOnceInitialize

RunOnceInitialization

RtlRunOnceInitialize function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlRunOnceInitialize** routine initializes a RTL_RUN_ONCE structure.

Syntax

```
NTSYSAPI VOID RtlRunOnceInitialize(
  PRTL_RUN_ONCE RunOnce
);
```

Parameters

`RunOnce`

Pointer to the **RTL_RUN_ONCE** one-time initialization structure.

Return Value

None

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

RTL_RUN_ONCE

RtlRunOnceBeginInitialize

RtlRunOnceComplete

RtlRunOnceExecuteOnce

RunOnceInitialization

RtlUpcaseUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUpcaseUnicodeString** routine converts a copy of the source string to uppercase and writes the converted string in the destination buffer.

Syntax

```
NTSYSAPI NTSTATUS RtlUpcaseUnicodeString(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  BOOLEAN          AllocateDestinationString
);
```

Parameters

`DestinationString`

Pointer to a caller-allocated buffer for the converted Unicode string.

`SourceString`

Pointer to the source Unicode string to be converted to uppercase.

`AllocateDestinationString`

Specifies if **RtlUpcaseUnicodeString** is to allocate the buffer space for the *DestinationString*. If it does, the buffer must be deallocated by calling **RtlFreeUnicodeString**.

Return Value

If the operation succeeds, **RtlUpcaseUnicodeString** returns STATUS_SUCCESS. Otherwise, no storage was allocated, and no conversion was done.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL"

See Also

RtlFreeUnicodeString

RtlUpcaseUnicodeChar

RtlUpperChar function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUpperChar** routine converts the specified character to uppercase.

Syntax

```
NTSYSAPI CHAR RtlUpperChar(
  CHAR Character
);
```

Parameters

`Character`

Specifies the character to convert.

Return Value

RtlUpperChar returns the uppercase version of the specified character or returns the value specified by the caller for *Character* if the specified character cannot be converted.

Remarks

RtlUpperChar returns the input *Character* unconverted if it is the lead byte of a multibyte character or if the uppercase equivalent of *Character* is a double-byte character. To convert such characters, use **RtlUpcaseUnicodeChar**.

RtlUpperChar is not recommended because the function is designed to work for ASCII range A-Z letters. Instead, use **RtlUpcaseUnicodeChar**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

RtlUpcaseUnicodeChar

RtlUpperString

RtlUpperString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUpperString** routine copies the given *SourceString* to the *DestinationString* buffer, converting it to uppercase.

Syntax

```
NTSYSAPI VOID RtlUpperString(
  PSTRING      DestinationString,
  const STRING *SourceString
);
```

Parameters

`DestinationString`

Pointer to the buffer for the converted destination string.

`SourceString`

Pointer to the source string to be converted to uppercase.

Return Value

None

Remarks

The **MaximumLength** and **Buffer** members of *DestinationString* are not modified by this routine.

The number of bytes copied from *SourceString* is either the **Length** of *SourceString* or the **MaximumLength** of *DestinationString*, whichever is smaller.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

RtlUpperChar

RtlValidateCorrelationVector function

1/25/2019 • 2 minutes to read • Edit Online

Validates the specified correlation vector to check whether it conforms to the Correlation Vector Specification (v2). The function specifically checks if the first 22 bytes are a valid base64 representation of a 16 byte buffer and the remaining characters match the (.\d+)+ regular expression.

Syntax

```
NTSYSAPI NTSTATUS RtlValidateCorrelationVector(
  PCORRELATION_VECTOR Vector
);
```

Parameters

`Vector`

A pointer to a **CORRELATION_VECTOR** structure that represents the correlation vector to be validated.

Return Value

Returns an NTSTATUS value that indicates the success of failure of the operation.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The correlation vector is valid.
STATUS_INVALID_PARAMETER	The supplied correlation vector is invalid.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Target Platform	Windows
Header	ntddk.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

RtlVolumeDeviceToDosName function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlVolumeDeviceToDosName** routine is obsolete for Windows XP and later versions of Windows. Use IoVolumeDeviceToDosName instead.

RtlVolumeDeviceToDosName returns the MS-DOS path for a specified device object that represents a file system volume.

Syntax

```
NTSYSAPI NTSTATUS RtlVolumeDeviceToDosName(
  PVOID           VolumeDeviceObject,
  PUNICODE_STRING DosName
);
```

Parameters

`VolumeDeviceObject`

Pointer to a device object that represents a volume device object created by a storage class driver.

`DosName`

Pointer to a Unicode string containing the MS-DOS path of the volume device object specified by *VolumeDeviceObject*.

Return Value

RtlVolumeDeviceToDosName returns STATUS_SUCCESS or an appropriate error status.

Remarks

The behavior of this routine is identical to that of IoVolumeDeviceToDosName. For more information about how to use this routine, see **IoVolumeDeviceToDosName**.

Drivers that must work on older NT-based operating systems may use this routine. Drivers written for Windows XP and later must use **IoVolumeDeviceToDosName** instead.

Requirements

Minimum supported client	Obsolete for Microsoft Windows XP and later versions of Windows. Use IoVolumeDeviceToDosName instead.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe

SeSinglePrivilegeCheck function

1/25/2019 • 2 minutes to read • Edit Online

The **SeSinglePrivilegeCheck** routine checks for the passed privilege value in the context of the current thread.

Syntax

```
BOOLEAN SeSinglePrivilegeCheck(
  LUID            PrivilegeValue,
  KPROCESSOR_MODE PreviousMode
);
```

Parameters

`PrivilegeValue`

Specifies the LUID value of the privilege being checked.

`PreviousMode`

Specifies the previous execution mode, one of **UserMode** or **KernelMode**.

Return Value

SeSinglePrivilegeCheck returns **TRUE** if the current subject has the required privilege.

Remarks

If *PreviousMode* is **KernelMode**, the privilege check always succeeds. Otherwise, this routine uses the token of the user-mode thread to determine whether the current (user-mode) thread has been granted the given privilege.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

LUID

RtlConvertLongToLuid

RtlConvertUlongToLuid

RtlEqualLuid

SeValidSecurityDescriptor

SILO_CONTEXT_CLEANUP_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

This callback is invoked when the context object reaches a reference count of zero.

Syntax

```
SILO_CONTEXT_CLEANUP_CALLBACK SiloContextCleanupCallback;

void SiloContextCleanupCallback(
  PVOID SiloContext
)
{...}
```

Parameters

`SiloContext`

A pointer provided by the user with the call to PsCreateSiloContext.

Return Value

This callback function does not return a value.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

SILO_MONITOR_CREATE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

This is callback is invoked when a new silo is created.

Syntax

```
SILO_MONITOR_CREATE_CALLBACK SiloMonitorCreateCallback;

NTSTATUS SiloMonitorCreateCallback(
  PESILO Silo
)
{...}
```

Parameters

`Silo`

The silo that was created.

Return Value

The NT code returned by the operation.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

SILO_MONITOR_TERMINATE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

This callback is invoked when a silo is terminated.

Syntax

```
SILO_MONITOR_TERMINATE_CALLBACK SiloMonitorTerminateCallback;

void SiloMonitorTerminateCallback(
  PESILO Silo
)
{...}
```

Parameters

`Silo`

The silo to be terminated.

Return Value

This callback function does not return a value.

Remarks

The expected behavior is that the component will drop any outstanding silo references. A driver may no longer operate within the namespace of a silo (via PsAttachSiloToCurrentThread) once it has returned from this function.

Requirements

Minimum supported client	Windows 10, version 1607
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ntddk.h

ZwAllocateLocallyUniqueId function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwAllocateLocallyUniqueId** routine allocates a locally unique identifier (LUID).

Syntax

```
NTSYSAPI NTSTATUS ZwAllocateLocallyUniqueId(
  PLUID Luid
);
```

Parameters

`Luid`

Pointer to a caller-allocated buffer to receive a copy of the source LUID structure. The buffer must be at least **sizeof**(*LUID*).

Return Value

ZwAllocateLocallyUniqueId returns STATUS_SUCCESS if the operation succeeds or the appropriate NTSTATUS value if the operation fails. If the output parameter cannot be written to, the routine returns STATUS_ACCESS_VIOLATION.

Remarks

The allocated LUID is unique to the local system only, and uniqueness is guaranteed only until the system is next restarted.

> **Note** If the call to the **ZwAllocateLocallyUniqueId** function occurs in user mode, you should use the name "**NtAllocateLocallyUniqueId**" instead of "**ZwAllocateLocallyUniqueId**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

LUID

Using Nt and Zw Versions of the Native System Services Routines

ZwDeviceIoControlFile function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwDeviceIoControlFile** routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.

Syntax

```
NTSYSAPI NTSTATUS ZwDeviceIoControlFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  ULONG            IoControlCode,
  PVOID            InputBuffer,
  ULONG            InputBufferLength,
  PVOID            OutputBuffer,
  ULONG            OutputBufferLength
);
```

Parameters

`FileHandle`

Handle returned by ZwCreateFile or ZwOpenFile for the file object representing the device to which the control information should be given or from which information should be returned. The file object must have been opened for asynchronous I/O if the caller specifies an *Event*, *ApcRoutine*, and an APC context (in *ApcContext*), or a completion context (in *ApcContext*). For I/O to an underlying mass-storage device, the file object must have been opened for Direct Access to Storage Device (DASD) access.

`Event`

Handle for a caller-created event. If this parameter is supplied, the caller will be put into a wait state until the requested operation is completed and the given event is set to the Signaled state. This parameter is optional and can be **NULL**. It must be **NULL** if the caller will wait for the *FileHandle* to be set to the Signaled state.

`ApcRoutine`

Address of an optional, caller-supplied APC routine to be called when the requested operation completes. This parameter can be **NULL**. It must be **NULL** if there is an I/O completion object associated with the file object.

`ApcContext`

Pointer to a caller-determined context area. This parameter value is used as the APC context if the caller supplies an APC, or is used as the completion context if an I/O completion object has been associated with the file object. When the operation completes, either the APC context is passed to the APC, if one was specified, or the completion context is included as part of the completion message that the I/O Manager posts to the associated I/O completion object.

This parameter is optional and can be **NULL**. It must be **NULL** if *ApcRoutine* is **NULL** and there is no I/O completion object associated with the file object.

`IoStatusBlock`

Pointer to a variable that receives the final completion status and information about the operation. For successful calls that return data, the number of bytes written to the *OutputBuffer* is returned in the **Information** member.

`IoControlCode`

IOCTL_*XXX* code that indicates which device I/O control operation is to be carried out on, usually by the underlying device driver. The value of this parameter determines the format and required length of the *InputBuffer* and *OutputBuffer*, as well as which of the following parameter pairs are required. For detailed information about the system-defined, device-type-specific IOCTL_*XXX* codes, see the device technology-specific section of the Microsoft Windows Driver Kit (WDK) documentation and Device Input and Output Control Codes in the Microsoft Windows SDK documentation.

`InputBuffer`

Pointer to a caller-allocated input buffer that contains device-specific information to be given to the target device. If *IoControlCode* specifies an operation that does not require input data, this pointer can be **NULL**.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. If *InputBuffer* is **NULL**, set *InputBufferLength* to zero.

`OutputBuffer`

Pointer to a caller-allocated output buffer in which information is returned from the target device. If *IoControlCode* specifies an operation that does not produce output data, this pointer can be **NULL**.

`OutputBufferLength`

Size, in bytes, of the buffer at *OutputBuffer*. If *OutputBuffer* is **NULL**, set *OutputBufferLength* to zero.

Return Value

ZwDeviceIoControlFile returns STATUS_SUCCESS if the underlying driver(s) successfully carried out the requested operation. Otherwise, the return value can be an error status code propagated from an underlying driver. Possible error status codes include the following:

Remarks

ZwDeviceIoControlFile provides a consistent view of the input and output data to the system and to kernel-mode drivers, while providing applications and underlying drivers with a device-dependent method of specifying a communications interface.

For more information about system-defined IOCTL_*XXX* codes, and about defining driver-specific IOCTL_*XXX* or FSCTL_*XXX* values, see Using I/O Control Codes in the *Kernel Mode Architecture Guide* and Device Input and Output Control Codes in the Microsoft Windows SDK documentation.

If the caller opened the file for asynchronous I/O (with neither FILE_SYNCHRONOUS_*XXX* create/open option set), the specified event, if any, will be set to the signaled state when the device control operation completes. Otherwise, the file object specified by *FileHandle* will be set to the signaled state. If an *ApcRoutine* was specified, it is called with the *ApcContext* and *IoStatusBlock* pointers.

Minifilters should use FltDeviceIoControlFile instead of **ZwDeviceIoControlFile**.

Callers of **ZwDeviceIoControlFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **ZwDeviceIoControlFile** function occurs in user mode, you should use the name "**NtDeviceIoControlFile**" instead of "**ZwDeviceIoControlFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntifs.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltDeviceIoControlFile

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoCallDriver

Using I/O Control Codes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateFile

ZwOpenFile

ZwOpenProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenProcess** routine opens a handle to a process object and sets the access rights to this object.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenProcess(
  PHANDLE            ProcessHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PCLIENT_ID         ClientId
);
```

Parameters

`ProcessHandle`

A pointer to a variable of type HANDLE. The **ZwOpenProcess** routine writes the process handle to the variable that this parameter points to.

`DesiredAccess`

An ACCESS_MASK value that contains the access rights that the caller has requested to the process object.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the attributes to apply to the process object handle. In Windows Vista and later versions of Windows, the **ObjectName** field of this structure must be set to **NULL**. In Windows Server 2003, Windows XP, and Windows 2000, this field can, as an option, point to an object name. For more information, see the following Remarks section.

`ClientId`

A pointer to a client ID that identifies the thread whose process is to be opened. In Windows Vista and later versions of Windows, this parameter must be a non-**NULL** pointer to a valid client ID. In Windows Server 2003, Windows XP, and Windows 2000, this parameter is optional and can be set to **NULL** if the **OBJECT_ATTRIBUTES** structure that *ObjectAttributes* points to specifies an object name. For more information, see the following Remarks section.

Return Value

ZwOpenProcess returns STATUS_SUCCESS if the call is successful. Possible return values include the following error status codes.

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER_MIX	In Windows Vista and later versions of Windows, the caller either supplied an object name or failed to supply a client ID. In Windows Server 2003, Windows XP, and Windows 2000, the caller supplied both an object name and a client ID.
STATUS_INVALID_CID	The specified client ID is not valid.
STATUS_INVALID_PARAMETER	The requested access rights are not valid for a process object.
STATUS_ACCESS_DENIED	The requested access rights cannot be granted.

Remarks

In Windows Vista and later versions of Windows, the *ClientId* parameter must point to a client ID that identifies the thread whose process is to be opened. In addition, the **ObjectName** field of the structure pointed to by *ObjectAttributes* must be set to **NULL**.

In Windows Server 2003, Windows XP, and Windows 2000, the caller has the option of supplying either a client ID or an object name (but not both). If the **ObjectName** field of the structure pointed to by *ObjectAttributes* contains a non-**NULL** pointer to an object name, *ClientId* must be **NULL**.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenProcess**" instead of "**ZwOpenProcess**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwPowerInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPowerInformation** routine sets or retrieves system power information.

Syntax

```
NTSYSAPI NTSTATUS ZwPowerInformation(
  POWER_INFORMATION_LEVEL InformationLevel,
  PVOID                   InputBuffer,
  ULONG                   InputBufferLength,
  PVOID                   OutputBuffer,
  ULONG                   OutputBufferLength
);
```

Parameters

`InformationLevel`

Specifies the requested information level, which indicates the specific power information to be set or retrieved. Currently, the only supported *POWER_INFORMATION_LEVEL* value is **PlatformInformation**.

VALUE	MEANING
PlatformInformation	Information represents the currently supported power capabilities of the system. Information may change as drivers are installed. For example, the installation of legacy device drivers that do not support power management might modify the capabilities of the system.

`InputBuffer`

Pointer to a caller-allocated input buffer. This parameter must be **NULL**, otherwise **ERROR_INVALID_PARAMETER** is returned.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. The parameter must be set to zero.

`OutputBuffer`

A pointer to an output buffer. The data type of this buffer depends on the information level requested in the *InformationLevel* parameter. For the **PlatformInformation** level, the only currently supported value, the *OutputBuffer* parameter is required and should be of the POWER_PLATFORM_INFORMATION type.

`OutputBufferLength`

Size, in bytes, of the output buffer. Depending on the information level requested, the buffer may be variably sized. *PlatformInformation*, the only currently supported value, requires a buffer that is the size of a POWER_PLATFORM_INFORMATION structure.

Return Value

Returns STATUS_SUCCESS if the call is successful. If the call fails, possible error codes include the following:

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The output buffer is of insufficient size to contain the data being returned.
STATUS_INVALID_PARAMETER	The **PlatformInformation** information level, which is the only currently supported value, requires no input buffer and must contain an output buffer. The caller either supplied an input buffer or no output buffer.
STATUS_ACCESS_DENIED	The caller had insufficient access rights to perform the requested action.

Remarks

NtPowerInformation and **ZwPowerInformation** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Examples

This example illustrates a valid function call.

C++

```
POWER_PLATFORM_INFORMATION PlatformInfo = {0};
NTSTATUS Result = NtPowerInformation(PlatformInformation, NULL, 0, &PlatformInfo, sizeof(PlatformInfo));
```

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Target Platform	Universal
Header	ntddk.h (include Wdm.h, Ntddk.h, Ntpoapi.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

POWER_PLATFORM_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwQueryVolumeInformationFile function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwQueryVolumeInformationFile** routine retrieves information about the volume associated with a given file, directory, storage device, or volume.

Syntax

```
NTSYSAPI NTSTATUS ZwQueryVolumeInformationFile(
  HANDLE               FileHandle,
  PIO_STATUS_BLOCK     IoStatusBlock,
  PVOID                FsInformation,
  ULONG                Length,
  FS_INFORMATION_CLASS FsInformationClass
);
```

Parameters

`FileHandle`

A handle to a file object returned by ZwCreateFile or ZwOpenFile for an open file, directory, storage device, or volume for which volume information is being requested.

`IoStatusBlock`

A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the query operation. For successful calls that return data, the number of bytes written to the *FsInformation* buffer is returned in the structure's **Information** member.

`FsInformation`

A pointer to a caller-allocated buffer that receives the desired information about the volume. The structure of the information returned in the buffer is defined by the *FsInformationClass* parameter.

`Length`

Size in bytes of the buffer pointed to by *FsInformation*. The caller should set this parameter according to the given *FsInformationClass*.

`FsInformationClass`

Type of information to be returned about the volume. Set this member to one of the following FS_INFORMATION_CLASS enumeration values.

VALUE	MEANING
FileFsAttributeInformation	Return a FILE_FS_ATTRIBUTE_INFORMATION structure containing attribute information about the file system responsible for the volume.
FileFsControlInformation	Return a FILE_FS_CONTROL_INFORMATION structure containing file system control information about the volume.

FileFsDeviceInformation	Return a FILE_FS_DEVICE_INFORMATION structure containing device information for the volume.
FileFsDriverPathInformation	Return a FILE_FS_DRIVER_PATH_INFORMATION structure containing information about whether a specified driver is in the I/O path for the volume. The caller must store the name of the driver into the **FILE_FS_DRIVER_PATH_INFORMATION** structure before calling **ZwQueryVolumeInformationFile**.
FileFsFullSizeInformation	Return a FILE_FS_FULL_SIZE_INFORMATION structure containing information about the total amount of space available on the volume.
FileFsObjectIdInformation	Return a FILE_FS_OBJECTID_INFORMATION structure containing file system-specific object ID information for the volume. Note that this is not the same as the (GUID-based) unique volume name assigned by the operating system.
FileFsSizeInformation	Return a FILE_FS_SIZE_INFORMATION structure containing information about the amount of space on the volume that is available to the user associated with the calling thread.
FileFsVolumeInformation	Return a FILE_FS_VOLUME_INFORMATION containing information about the volume such as the volume label, serial number, and creation time.
FileFsSectorSizeInformation	Return a FILE_FS_SECTOR_SIZE_INFORMATION structure that contains information about the physical and logical sector sizes of a volume.

Return Value

ZwQueryVolumeInformationFile returns STATUS_SUCCESS or an appropriate error status.

Remarks

ZwQueryVolumeInformationFile retrieves information about the volume associated with a given file, directory, storage device, or volume.

If the *FileHandle* represents a direct device open, only *FileFsDeviceInformation* can be specified as the value of *FsInformationClass*.

ZwQueryVolumeInformationFile returns zero in any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by the file system.

For information about other file information query routines, see File Objects.

Minifilters should use FltQueryVolumeInformationFile instead of **ZwQueryVolumeInformationFile**.

Callers of **ZwQueryVolumeInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **ZwQueryVolumeInformationFile** function occurs in user mode, you should use the name "**NtQueryVolumeInformationFile**" instead of "**ZwQueryVolumeInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntddk.h (include Ntifs.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_FS_ATTRIBUTE_INFORMATION

FILE_FS_CONTROL_INFORMATION

FILE_FS_DEVICE_INFORMATION

FILE_FS_DRIVER_PATH_INFORMATION

FILE_FS_FULL_SIZE_INFORMATION

FILE_FS_OBJECTID_INFORMATION

FILE_FS_SIZE_INFORMATION

FILE_FS_VOLUME_INFORMATION

FltQueryVolumeInformationFile

IRP_MJ_QUERY_VOLUME_INFORMATION

IRP_MJ_SET_VOLUME_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateFile

ZwOpenFile

ZwQueryDirectoryFile

ZwQueryInformationFile

ZwSetInformationFile

ZwSetVolumeInformationFile

ZwSetInformationThread function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationThread** routine sets the priority of a thread.

Syntax

```
NTSYSAPI NTSTATUS ZwSetInformationThread(
  HANDLE          ThreadHandle,
  THREADINFOCLASS ThreadInformationClass,
  PVOID           ThreadInformation,
  ULONG           ThreadInformationLength
);
```

Parameters

`ThreadHandle`

Handle to the thread object. To create a new thread and get a handle to it, call PsCreateSystemThread. To specify the current thread, use the ZwCurrentThread macro.

`ThreadInformationClass`

One of the system-defined values in the THREADINFOCLASS enumeration (see ntddk.h), **ThreadPriority**, **ThreadBasePriority**, **ThreadPagePriority**, or **ThreadPowerThrottlingState**.

`ThreadInformation`

Pointer to a variable that specifies the information to set.

If **ThreadInformationClass** is **ThreadPriority**, this value must be > LOW_PRIORITY and <= HIGH_PRIORITY.

If **ThreadInformationClass** is **ThreadBasePriority**, this value must fall within the system's valid base-priority range and the original priority class for the given thread. That is, if a thread's priority class is variable, that thread's base priority cannot be reset to a real-time priority value, and vice versa.

If **ThreadInformationClass** is **ThreadPagePriority**, this value is a pointer to a **PAGE_PRIORITY_INFORMATION** structure, see ntddk.h. The **PagePriority** member value must be one of these values.

If **ThreadInformationClass** is **ThreadPowerThrottlingState**, this value is a pointer to a POWER_THROTTLING_THREAD_STATE structure, see ntddk.h. The **PagePriority** member value must be one of these values.

VALUE	MEANING

MEMORY_PRIORITY_VERY_LOW 1	Very low memory priority.
MEMORY_PRIORITY_LOW 2	Low memory priority.
MEMORY_PRIORITY_MEDIUM 3	Medium memory priority.
MEMORY_PRIORITY_BELOW_NORMAL 4	Below normal memory priority.
MEMORY_PRIORITY_NORMAL 5	Normal memory priority. This is the default priority for all threads and processes on the system.

`ThreadInformationLength`

The size, in bytes, of **ThreadInformation**.

Return Value

ZwSetInformationThread returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. Possible error codes include STATUS_INFO_LENGTH_MISMATCH or STATUS_INVALID_PARAMETER.

Remarks

ZwSetInformationThread can be called by higher-level drivers to set the priority of a thread for which they have a handle.

The caller must have THREAD_SET_INFORMATION access rights for the given thread in order to call this routine.

Usually, device and intermediate drivers that set up driver-created threads call KeSetBasePriorityThread or KeSetPriorityThread from their driver-created threads, rather than calling **ZwSetInformationThread**. However, a driver can call **ZwSetInformationThread** to raise the priority of a driver-created thread before that thread runs.

Kernel mode drivers can call the **ZwSetInformationThread** function with **ThreadPagePriority** to specify a thread's page priority.

To help improve system performance, drivers should use the function with **ThreadPagePriority** to lower the page priority of threads that perform background operations or access files and data that are not expected to be accessed again soon. For example, an anti-malware application might lower the priority of threads involved in scanning files.

To determine the page priority for a thread, call ZwQueryInformationThread.

> **Note** If the call to this function occurs in user mode, you should use the name NtSetInformationThread instead of **ZwSetInformationThread**.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeSetBasePriorityThread

KeSetPriorityThread

PsCreateSystemThread

Using Nt and Zw Versions of the Native System Services Routines

ZwTerminateProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwTerminateProcess** routine terminates a process and all of its threads.

Syntax

```
NTSYSAPI NTSTATUS ZwTerminateProcess(
  HANDLE   ProcessHandle,
  NTSTATUS ExitStatus
);
```

Parameters

`ProcessHandle`

A handle to the process object that represents the process to be terminated.

`ExitStatus`

An NTSTATUS value that the operating system uses as the final status for the process and each of its threads.

Return Value

ZwTerminateProcess returns STATUS_SUCCESS if the operation succeeds. Additional return values include:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a process handle.
STATUS_INVALID_HANDLE	The specified handle is not valid.

STA TUS _AC CES S_D ENI ED	The driver cannot access the specified process object.
STA TUS _PR OCE SS_I S_T ER MIN ATI NG	The specified process is already terminating.

If the caller specifies the current process in the *ProcessHandle* parameter, **ZwTerminateProcess** does not return.

Remarks

To obtain a process handle that a driver can specify for the *ProcessHandle* parameter, the driver can call ZwOpenProcess. The handle must be a *kernel handle*, a handle that can only be accessed in kernel mode. A handle is a kernel handle if it is created with the OBJ_KERNEL_HANDLE flag. For more info see InitializeObjectAttributes.

Drivers must not specify the current process if resources have not been freed from the kernel stack, because the operating system will not unwind the kernel stack for the calling thread.

> **Note** If the call to this function occurs in user mode, you should use the name "NtTerminateProcess" instead of "**ZwTerminateProcess**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntddk.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwOpenProcess

ntddsfio.h header

1/8/2019 • 2 minutes to read • Edit Online

This is the include file that defines all common constants, ioctls and types for Scheduled File I/O (SFIO).

This header is used by kernel. For more information, see:

- Windows kernel ntddsfio.h contains the following programming interfaces:

Enumerations

TITLE	DESCRIPTION
IO_ACCESS_MODE	Defines the types of access mode for Scheduled File I/O (SFIO).
IO_ACCESS_TYPE	Defines the access rights for Scheduled File I/O (SFIO).

IO_ACCESS_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Defines the types of access mode for Scheduled File I/O (SFIO).

Syntax

```
typedef enum _IO_ACCESS_MODE {
  SequentialAccess,
  RandomAccess
} IO_ACCESS_MODE;
```

Constants

SequentialAccess	Indicates that the input/output will be sent down in a sequential order.
RandomAccess	Indicates that the input/output might not be in a predictable order.

Requirements

Header	ntddsfio.h (include Wdm.h, Ntddsfio.h)

IO_ACCESS_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Defines the access rights for Scheduled File I/O (SFIO).

Syntax

```
typedef enum _IO_ACCESS_TYPE {
  ReadAccess,
  WriteAccess,
  ModifyAccess
} IO_ACCESS_TYPE;
```

Constants

ReadAccess	Indicates that the input/output will be comprised solely of reads.
WriteAccess	Indicates that the input/output will be comprised solely of writes.
ModifyAccess	Indicates that the input/output will be comprised of reads and writes.

Requirements

Header	ntddsfio.h (include Ntddsfio.h)

ntddsysenv.h header

1/8/2019 • 2 minutes to read • Edit Online

This is the include file that defines all constants and types for accessing the system variable device interface.

- Windows kernel ntddsysenv.h contains the following programming interfaces:

IOCTLs

TITLE	DESCRIPTION
IOCTL_SYSENV_ENUM_VARIABLES	Returns information about system environment variables using SysEnv device.
IOCTL_SYSENV_GET_VARIABLE	Gets the value of the specified system environment variables using SysEnv device.
IOCTL_SYSENV_QUERY_VARIABLE_INFO	Returns information about system environment variables using SysEnv device.
IOCTL_SYSENV_SET_VARIABLE	Sets the value of the specified system environment variables using SysEnv device.

Structures

TITLE	DESCRIPTION
SYSENV_VALUE	Stores the value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.
SYSENV_VARIABLE	Stores the name a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.
SYSENV_VARIABLE_INFO	Stores the information about a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_QUERY_VARIABLE_INFO request.
XVARIABLE_NAME	Stores the name of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES request.
XVARIABLE_NAME_AND_VALUE	Stores the name and value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES and IOCTL_SYSENV_SET_VARIABLE requests.

SYSENV_VALUE structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.

Syntax

```
typedef struct _SYSENV_VALUE {
  ULONG Attributes;
  ULONG ValueLength;
  UCHAR ValueData[ANYSIZE_ARRAY];
} SYSENV_VALUE, *PSYSENV_VALUE;
```

Members

`Attributes`

Attributes of the system environment variable.

`ValueLength`

The length of the value of the system environment variable.

`ValueData`

Requirements

Header	ntddsysenv.h

See Also

IOCTL_SYSENV_GET_VARIABLE

SYSENV_VARIABLE structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the name a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_GET_VARIABLE request.

Syntax

```
typedef struct _SYSENV_VARIABLE {
  GUID  VendorGuid;
  ULONG VariableNameLength;
  WCHAR VariableName[ANYSIZE_ARRAY];
} SYSENV_VARIABLE, *PSYSENV_VARIABLE;
```

Members

`VendorGuid`

The vendor GUID.

`VariableNameLength`

The length of the string pointed to by **VariableName**, which contains the name of the variable.

`VariableName`

A string that contains the name of the variable.

Requirements

Header	ntddsysenv.h

See Also

IOCTL_SYSENV_GET_VARIABLE

SYSENV_VARIABLE_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the information about a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_QUERY_VARIABLE_INFO request.

Syntax

```
typedef struct _SYSENV_VARIABLE_INFO {
  ULONGLONG MaximumVariableStorageSize;
  ULONGLONG RemainingVariableStorageSize;
  ULONGLONG MaximumVariableSize;
} SYSENV_VARIABLE_INFO, *PSYSENV_VARIABLE_INFO;
```

Members

`MaximumVariableStorageSize`

The size of the variable.

`RemainingVariableStorageSize`

The remaining size of the variable.

`MaximumVariableSize`

The maximum size of the variable.

Requirements

Header	ntddsysenv.h

See Also

IOCTL_SYSENV_QUERY_VARIABLE_INFO

XVARIABLE_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the name of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES request.

Syntax

```
typedef struct _XVARIABLE_NAME {
  ULONG NextEntryOffset;
  GUID  VendorGuid;
  WCHAR Name[ANYSIZE_ARRAY];
} XVARIABLE_NAME, *PXVARIABLE_NAME;
```

Members

`NextEntryOffset`

The location of the next entry in the array of **XVARIABLE_NAME** structures reprieved by the IOCTL_SYSENV_ENUM_VARIABLES request.

`VendorGuid`

The vendor GUID.

`Name`

Name of the system environment variable.

Requirements

Header	ntddsysenv.h

See Also

IOCTL_SYSENV_ENUM_VARIABLES

XVARIABLE_NAME_AND_VALUE structure

1/30/2019 • 2 minutes to read • Edit Online

Stores the name and value of a system environment variable using SysEnv device. This structure is used in the IOCTL_SYSENV_ENUM_VARIABLES and IOCTL_SYSENV_SET_VARIABLE requests.

Syntax

```
typedef struct _XVARIABLE_NAME_AND_VALUE {
  ULONG NextEntryOffset;
  ULONG ValueOffset;
  ULONG ValueLength;
  ULONG Attributes;
  GUID  VendorGuid;
  WCHAR Name[ANYSIZE_ARRAY];
} XVARIABLE_NAME_AND_VALUE, *PXVARIABLE_NAME_AND_VALUE;
```

Members

`NextEntryOffset`

The location of the next entry in the array of XVARIABLE_NAME structures reprieved by the IOCTL_SYSENV_ENUM_VARIABLES request.

`ValueOffset`

The value of the system environment variable.

`ValueLength`

The length of the value.

`Attributes`

Attributes of the system environment variable.

`VendorGuid`

The vendor GUID.

`Name`

Name of the system environment variable.

Requirements

Header	ntddsysenv.h

See Also

IOCTL_SYSENV_ENUM_VARIABLES

IOCTL_SYSENV_ENUM_VARIABLES IOCTL

1/25/2019 • 2 minutes to read • Edit Online

Returns information about system environment variables using SysEnv device.

Major Code

IRP_MJ_DEVICE_CONTROL

Input Buffer

A boolean value indicating whether values should be included.

Input Buffer Length

Size of BOOLEAN.

Output Buffer

A pointer to a buffer that contains the system environment variable information. If values have been requested, the buffer points to an array of XVARIABLE_NAME_AND_VALUE structures. Otherwise, returns an array of XVARIABLE_NAME structures.

The location of the next entry is determined by **NextEntryOffset** of the XVARIABLE_NAME_AND_VALUE and XVARIABLE_NAME.

Output Buffer Length

On input, the length in bytes of the output buffer. On output, the length in bytes of the returned data. If the input buffer is large enough, then this value indicates the amount of data copied into output buffer. If the input buffer is too small, then it indicates the required buffer length.

Status Block

Irp->IoStatus.Status is set to STATUS_SUCCESS if the request is successful. Otherwise, **Status** to the appropriate error condition as a NTSTATUS code.

Requirements

Header	ntddsysenv.h

See Also

Creating IOCTL Requests in Drivers

WdfIoTargetSendInternalIoctlOthersSynchronously

WdfIoTargetSendInternalIoctlSynchronously

WdfIoTargetSendIoctlSynchronously

IOCTL_SYSENV_GET_VARIABLE IOCTL

1/25/2019 • 2 minutes to read • Edit Online

Gets the value of the specified system environment variables using SysEnv device.

Major Code

IRP_MJ_DEVICE_CONTROL

Input Buffer

A pointer to a SYSENV_VARIABLE structure that specifies the variable..

Input Buffer Length

The size of the SYSENV_VARIABLE structure.

Output Buffer

A pointer to a SYSENV_VALUE structure that receives the value of the variable.

Output Buffer Length

The size of the SYSENV_VALUE structure.

Status Block

Irp->IoStatus.Status is set to STATUS_SUCCESS if the request is successful. Otherwise, **Status** to the appropriate error condition as a NTSTATUS code.

Requirements

Header	ntddsysenv.h

See Also

Creating IOCTL Requests in Drivers

WdfIoTargetSendInternalIoctlOthersSynchronously

WdfIoTargetSendInternalIoctlSynchronously

WdfIoTargetSendIoctlSynchronously

IOCTL_SYSENV_QUERY_VARIABLE_INFO IOCTL

1/25/2019 • 2 minutes to read • Edit Online

Returns information about system environment variables using SysEnv device.

Major Code

IRP_MJ_DEVICE_CONTROL

Input Buffer

A pointer to a ULONG variable that contains the attributes of the system environment variable.

Input Buffer Length

Size of ULONG.

Output Buffer

A pointer to a SYSENV_VARIABLE_INFO structure that contains variable information.

Output Buffer Length

The size of the SYSENV_VARIABLE_INFO structure.

Status Block

Irp->IoStatus.Status is set to STATUS_SUCCESS if the request is successful. Otherwise, **Status** to the appropriate error condition as a NTSTATUS code.

Requirements

Header	ntddsysenv.h

See Also

Creating IOCTL Requests in Drivers

WdfIoTargetSendInternalIoctlOthersSynchronously

WdfIoTargetSendInternalIoctlSynchronously

WdfIoTargetSendIoctlSynchronously

IOCTL_SYSENV_SET_VARIABLE IOCTL

1/25/2019 • 2 minutes to read • Edit Online

Sets the value of the specified system environment variables using SysEnv device.

Major Code

IRP_MJ_DEVICE_CONTROL

Input Buffer

A pointer to a XVARIABLE_NAME_AND_VALUE structure that contains the information to set.

Input Buffer Length

The size of the XVARIABLE_NAME_AND_VALUE structure.

Status Block

Irp->IoStatus.Status is set to STATUS_SUCCESS if the request is successful. Otherwise, **Status** to the appropriate error condition as a NTSTATUS code.

Requirements

Header	ntddsysenv.h

See Also

Creating IOCTL Requests in Drivers

WdfIoTargetSendInternalIoctlOthersSynchronously

WdfIoTargetSendInternalIoctlSynchronously

WdfIoTargetSendIoctlSynchronously

ntifs.h header

1/8/2019 • 35 minutes to read • Edit Online

This header is used by ifsk. For more information, see:

- Installable file system ntifs.h contains the following programming interfaces:

IOCTLs

TITLE	DESCRIPTION
IOCTL_REDIR_QUERY_PATH	The IOCTL_REDIR_QUERY_PATH control code is sent by the multiple UNC provider (MUP) to network redirectors to determine which provider can handle a specific UNC path in a name-based operation, typically an IRP_MJ_CREATE request.
IOCTL_REDIR_QUERY_PATH_EX	The IOCTL_REDIR_QUERY_PATH_EX control code is sent by the multiple UNC provider (MUP) on Windows Vista or later to network redirectors to determine which provider can handle a specific UNC path in a name-based operation, typically an IRP_MJ_CREATE request.
IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES	The IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES control code is sent to force a flush of a file system before a volume shadow copy occurs.

Functions

TITLE	DESCRIPTION
CcCopyWriteWontFlush	The CcCopyWriteWontFlush macro determines whether the amount of data to be copied in a call to CcCopyWrite is small enough not to require immediate flushing to disk if CcCopyWrite is called with Wait set to FALSE.
ExAdjustLookasideDepth	Reserved for system use.
ExDisableResourceBoostLite	Reserved for system use.
ExQueryPoolBlockSize	Obsolete.
FsRtlAllocatePoolWithQuotaTag	The FsRtlAllocatePoolWithQuotaTag routine allocates pool memory, charging quota against the current process.
FsRtlAllocatePoolWithTag	The FsRtlAllocatePoolWithTag routine allocates pool memory.
FsRtlAreThereCurrentFileLocks	The FsRtlAreThereCurrentFileLocks macro checks whether any byte range locks exist for the specified file.
FsRtlCompleteRequest	The FsRtlCompleteRequest macro completes an IRP with the specified status.

TITLE	DESCRIPTION
FsRtlFastLock	The FsRtlFastLock macro is used by file systems and filter drivers to request a byte-range lock for a file stream.
FsRtlGetPerStreamContextPointer	The FsRtlGetPerStreamContextPointer macro returns the file system's stream context for a file stream.
FsRtlInitPerStreamContext	The FsRtlInitPerStreamContext macro initializes a filter driver context structure.
FsRtlIsAnsiCharacterLegal	The FsRtlIsAnsiCharacterLegal macro determines whether a character is a legal ANSI character.
FsRtlIsAnsiCharacterLegalFat	The FsRtlIsAnsiCharacterLegalFat macro determines whether an ANSI character is legal for FAT file names.
FsRtlIsAnsiCharacterLegalHpfs	The FsRtlIsAnsiCharacterLegalHpfs macro determines whether an ANSI character is legal for HPFS file names.
FsRtlIsAnsiCharacterLegalNtfs	The FsRtlIsAnsiCharacterLegalNtfs macro determines whether an ANSI character is legal for NTFS file names.
FsRtlIsAnsiCharacterLegalNtfsStream	The FsRtlIsAnsiCharacterLegalNtfsStream macro determines whether an ANSI character is legal for NTFS stream names.
FsRtlIsAnsiCharacterWild	The FsRtlIsAnsiCharacterWild macro determines whether an ANSI character is a wildcard character.
FsRtlIsLeadDbcsCharacter	The FsRtlIsLeadDbcsCharacter macro determines whether a character is a lead byte (the first byte of a character) in a double-byte character set (DBCS).
FsRtlIsUnicodeCharacterWild	The FsRtlIsUnicodeCharacterWild macro determines whether a Unicode character is a wildcard character.
FsRtlLookupPerStreamContext	The FsRtlLookupPerStreamContext macro retrieves a per-stream context structure for a file stream.
FsRtlRegisterFileSystemFilterCallbacks	File system filter drivers and file systems call the FsRtlRegisterFileSystemFilterCallbacks routine to register notification callback routines to be invoked when the underlying file system performs certain operations.
FsRtlSetupAdvancedHeaderEx	The FsRtlSetupAdvancedHeaderEx macro is used by file systems to initialize an FSRTL_ADVANCED_FCB_HEADER structure for use with both stream and file contexts.
FsRtlSupportsPerFileContexts	The FsRtlSupportsPerFileContexts macro checks if per file context information is supported by the file system that is associated with a specified FILE_OBJECT.
FsRtlTestAnsiCharacter	The FsRtlTestAnsiCharacter macro determines whether an ANSI or double-byte character set (DBCS) character meets the specified criteria.

TITLE	DESCRIPTION
IoAcquireVpbSpinLock	The IoAcquireVpbSpinLock routine acquires the Volume Parameter Block (VPB) spin lock.
IoCheckDesiredAccess	Reserved for system use.
IoCheckEaBufferValidity	The IoCheckEaBufferValidity routine checks whether the specified extended attribute (EA) buffer is valid.
IoCheckFunctionAccess	Reserved for system use.
IoCheckQuerySetFileInformation	Reserved for system use.
IoCheckQuerySetVolumeInformation	Reserved for system use.
IoCheckQuotaBufferValidity	The IoCheckQuotaBufferValidity routine checks whether the specified quota buffer is valid.
IoCreateStreamFileObject	The IoCreateStreamFileObject routine creates a new stream file object.
IoCreateStreamFileObjectEx	The IoCreateStreamFileObjectEx routine creates a new stream file object.
IoCreateStreamFileObjectEx2	The IoCreateStreamFileObjectEx2 routine creates a new stream file object with create options for a target device object.
IoCreateStreamFileObjectLite	The IoCreateStreamFileObjectLite routine creates a new stream file object, but does not cause an IRP_MJ_CLEANUP request to be sent to the file system driver stack.
IoEnumerateDeviceObjectList	The IoEnumerateDeviceObjectList routine enumerates a driver's device object list.
IoEnumerateRegisteredFiltersList	The IoEnumerateRegisteredFiltersList routine enumerates the file system filter drivers that have registered with the system.
IoFastQueryNetworkAttributes	Reserved for system use.
IoGetAttachedDevice	The IoGetAttachedDevice routine returns a pointer to the highest-level device object associated with the specified device.
IoGetAttachedDeviceReference	The IoGetAttachedDeviceReference routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.
IoGetBaseFileSystemDeviceObject	Reserved for system use.
IoGetConfigurationInformation	The IoGetConfigurationInformation routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.

TITLE	DESCRIPTION
IoGetDeviceAttachmentBaseRef	The IoGetDeviceAttachmentBaseRef routine returns a pointer to the lowest-level device object in a file system or device driver stack.
IoGetDeviceToVerify	The IoGetDeviceToVerify routine returns a pointer to the device object, representing a removable-media device, that is the target of the given thread's I/O request.
IoGetDiskDeviceObject	The IoGetDiskDeviceObject routine retrieves a pointer to the disk device object associated with a given file system volume device object.
IoGetLowerDeviceObject	The IoGetLowerDeviceObject routine returns a pointer to the next-lower-level device object on the driver stack.
IoGetRequestorProcess	The IoGetRequestorProcess routine returns a process pointer for the thread that originally requested a given I/O operation.
IoGetRequestorProcessId	The IoGetRequestorProcessId routine returns the unique 32-bit process ID for the thread that originally requested a given I/O operation.
IoGetRequestorSessionId	The IoGetRequestorSessionId routine returns the session ID for the process that originally requested a given I/O operation.
IoGetTopLevelIrp	The IoGetTopLevelIrp routine returns the value of the TopLevelIrp field of the current thread.
IoInitializePriorityInfo	The IoInitializePriorityInfo routine initializes a structure of type IO_PRIORITY_INFO.
IoIsFileOpenedExclusively	Reserved for system use.
IoIsOperationSynchronous	The IoIsOperationSynchronous routine determines whether a given IRP represents a synchronous or asynchronous I/O request.
IoIsSystemThread	The IoIsSystemThread routine checks whether a given thread is a system thread.
IoIsValidNameGraftingBuffer	Reserved for system use.
IoPageRead	Reserved for system use.
IoQueryFileDosDeviceName	The IoQueryFileDosDeviceName routine retrieves an MS-DOS device name for a file.
IoQueryFileInformation	Reserved for system use.
IoQueryVolumeInformation	Reserved for system use.
IoQueueThreadIrp	Reserved for system use.

TITLE	DESCRIPTION
IoRegisterFileSystem	The IoRegisterFileSystem routine adds a file system's control device object to the global file system queue.
IoRegisterFsRegistrationChange	The IoRegisterFsRegistrationChange routine registers a file system filter driver's notification routine to be called whenever a file system registers or unregisters itself as an active file system.
IoRegisterFsRegistrationChangeEx	The IoRegisterFsRegistrationChangeEx routine registers a file system filter driver's notification routine to be called whenever a file system registers or unregisters itself as an active file system.
IoRegisterFsRegistrationChangeMountAware	The IoRegisterFsRegistrationChangeMountAware routine registers a file system filter driver's notification routine. This notification routine is called whenever a file system registers or unregisters itself as an active file system.
IoReleaseVpbSpinLock	The IoReleaseVpbSpinLock routine releases the Volume Parameter Block (VPB) spin lock.
IoReplaceFileObjectName	The IoReplaceFileObjectName routine replaces the name of a file object.
IoSetDeviceToVerify	The IoSetDeviceToVerify routine specifies a device object to be verified. The specified device object represents a removable media device.
IoSetInformation	Reserved for system use.
IoSetStartIoAttributes	The IoSetStartIoAttributes routine sets attributes for the driver's StartIo routine.
IoSetTopLevelIrp	The IoSetTopLevelIrp routine sets the value of the TopLevelIrp field of the current thread.
IoSizeOfIrp	The IoSizeOfIrp routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.
IoStartNextPacket	The IoStartNextPacket routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.
IoStartNextPacketByKey	The IoStartNextPacketByKey routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.
IoStartPacket	The IoStartPacket routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.

TITLE	DESCRIPTION
IoStartTimer	The IoStartTimer routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.
IoStopTimer	The IoStopTimer routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.
IoSynchronousPageWrite	Reserved for system use.
IoThreadToProcess	The IoThreadToProcess routine returns a pointer to the process for the specified thread.
IoUnregisterFileSystem	The IoUnregisterFileSystem routine removes a file system's control device object from the global file system queue.
IoUnregisterFsRegistrationChange	The IoUnregisterFsRegistrationChange routine unregisters file system filter driver's file system registration change notification routine.
IoVerifyVolume	The IoVerifyVolume routine sends a volume verify request to the given removable-media device.
IoWriteErrorLogEntry	The IoWriteErrorLogEntry routine queues a given error log packet to the system error logging thread.
IsReparseTagMicrosoft	The IsReparseTagMicrosoft macro determines whether a reparse point tag indicates a Microsoft reparse point.
IsReparseTagNameSurrogate	The IsReparseTagNameSurrogate macro determines whether a tag's associated reparse point is a surrogate for another named entity, such as a volume mount point.
IsReparseTagValid	Reserved for system use.
KeAttachProcess	Obsolete.
KeDetachProcess	Obsolete.
KeGetProcessorIndexFromNumber	The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.
KeGetProcessorNumberFromIndex	The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.
KeInitializeMutant	Reserved for system use.
KeInitializeQueue	The KeInitializeQueue routine initializes a queue object on which threads can wait for entries.

TITLE	DESCRIPTION
KeInsertHeadQueue	The KeInsertHeadQueue routine inserts an entry at the head of the given queue if it cannot immediately use the entry to satisfy a thread wait.
KeInsertQueue	The KeInsertQueue routine inserts an entry at the tail of the given queue if it cannot immediately use the entry to satisfy a thread wait.
KeQueryPerformanceCounter	The KeQueryPerformanceCounter routine retrieves the current value and frequency of the performance counter.Use KeQueryPerformanceCounter to acquire high resolution (<1us) time stamps for time interval measurements.
KeReadStateMutant	Reserved for system use.
KeReadStateQueue	Reserved for system use.
KeReleaseMutant	Reserved for system use.
KeReleaseQueuedSpinLock	Reserved for system use.
KeRemoveQueue	The KeRemoveQueue routine gives the calling thread a pointer to a dequeued entry from the given queue object or allows the caller to wait, up to an optional timeout interval, on the queue object.
KeRundownQueue	The KeRundownQueue routine cleans up a queue object, flushing any queued entries.
KeSetIdealProcessorThread	Reserved for system use.
KeSetKernelStackSwapEnable	The KeSetKernelStackSwapEnable routine enables and disables swapping of the caller's stack to disk.
KeStackAttachProcess	The KeStackAttachProcess routine attaches the current thread to the address space of the target process.
KeStallExecutionProcessor	The KeStallExecutionProcessor routine stalls the caller on the current processor for a specified time interval.
KeTryToAcquireQueuedSpinLock	Reserved for system use.
KeUnstackDetachProcess	The KeUnstackDetachProcess routine detaches the current thread from the address space of a process and restores the previous attach state.
MmCanFileBeTruncated	The MmCanFileBeTruncated routine checks whether a file can be truncated.
MmDoesFileHaveUserWritableReferences	The MmDoesFileHaveUserWritableReferences function returns the number of writable references for a file object.
MmFlushImageSection	The MmFlushImageSection routine flushes the image section for a file.

TITLE	DESCRIPTION
MmForceSectionClosed	The MmForceSectionClosed routine deletes the data and image sections for a file that is no longer in use.
MmForceSectionClosedEx	The MmForceSectionClosedEx function examines the section object pointers. If they are NULL, no further action is taken and the value TRUE is returned.
MmGetMaximumFileSectionSize	The MmGetMaximumFileSectionSize returns the maximum possible size of a file section for the current version of Windows.
MmIsRecursiveIoFault	The MmIsRecursiveIoFault routine determines whether the current page fault is occurring during an I/O operation.
MmPrefetchPages	The MmPrefetchPages routine reads groups of pages from secondary storage in the optimal fashion.
MmSetAddressRangeModified	The MmSetAddressRangeModified routine marks currently valid pages in the specified range of the system cache as modified.
NtAllocateVirtualMemory	The ZwAllocateVirtualMemory routine reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process.
NtClose	The ZwClose routine closes an object handle.
NtCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
NtCreateSection	The ZwCreateSection routine creates a section object.
NtCreateSectionEx	Creates a section object.
NtDeviceIoControlFile	The ZwDeviceIoControlFile routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.
NtDuplicateToken	The ZwDuplicateToken function creates a handle to a new access token that duplicates an existing token. This function can create either a primary token or an impersonation token.
NtFlushBuffersFileEx	The ZwFlushBuffersFileEx routine is called by a file system filter driver to send a flush request for a given file to the file system. An optional flush operation flag can be set to control how file data is written to storage.
NtFreeVirtualMemory	The ZwFreeVirtualMemory routine releases, decommits, or both, a region of pages within the virtual address space of a specified process.
NtFsControlFile	The ZwFsControlFile routine sends a control code directly to a specified file system or file system filter driver, causing the corresponding driver to perform the specified action.

TITLE	DESCRIPTION
NtLockFile	The NtLockFile routine requests a byte-range lock for the specified file.
NtOpenFile	The ZwOpenFile routine opens an existing file, directory, device, or volume.
NtOpenProcessTokenEx	The ZwOpenProcessTokenEx routine opens the access token associated with a process.
NtOpenThreadTokenEx	The ZwOpenThreadTokenEx routine opens the access token associated with a thread.
NtQueryDirectoryFile	The ZwQueryDirectoryFile routine returns various kinds of information about files in the directory specified by a given file handle.
NtQueryInformationFile	The ZwQueryInformationFile routine returns various kinds of information about a file object.
NtQueryInformationToken	The ZwQueryInformationToken routine retrieves a specified type of information about an access token. The calling process must have appropriate access rights to obtain the information.
NtQueryObject	The ZwQueryObject routine provides information about a supplied object.
NtQueryQuotaInformationFile	The ZwQueryQuotaInformationFile routine retrieves quota entries associated with the volume specified by the FileHandle parameter.
NtQuerySecurityObject	The ZwQuerySecurityObject routine retrieves a copy of an object's security descriptor.
NtQueryVirtualMemory	The ZwQueryVirtualMemory routine determines the state, protection, and type of a region of pages within the virtual address space of the subject process.
NtQueryVolumeInformationFile	The ZwQueryVolumeInformationFile routine retrieves information about the volume associated with a given file, directory, storage device, or volume.
NtReadFile	The ZwReadFile routine reads data from an open file.
NtSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
NtSetInformationThread	The ZwSetInformationThread routine sets the priority of a thread.
NtSetInformationToken	The ZwSetInformationToken routine modifies information in a specified token. The calling process must have appropriate access rights to set the information.

TITLE	DESCRIPTION
NtSetQuotaInformationFile	The ZwSetQuotaInformationFile routine changes quota entries for the volume associated with the FileHandle parameter. All of the quota entries in the specified buffer are applied to the volume.
NtSetSecurityObject	The ZwSetSecurityObject routine sets an object's security state.
NtUnlockFile	The ZwUnlockFile routine unlocks a byte-range lock in a file.
NtWriteFile	The ZwWriteFile routine writes data to an open file.
ObInsertObject	Reserved for system use.
ObIsKernelHandle	The ObIsKernelHandle routine determines whether the specified handle is a kernel handle.
ObMakeTemporaryObject	Reserved for system use.
ObOpenObjectByPointer	The ObOpenObjectByPointer function opens an object referenced by a pointer and returns a handle to the object.
ObQueryNameString	The ObQueryNameString routine supplies the name, if there is one, of a given object to which the caller has a pointer.
ObQueryObjectAuditingByHandle	Reserved for system use.
PoCallDriver	The PoCallDriver routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoClearPowerRequest	The PoClearPowerRequest routine decrements the count for the specified power request type.
PoCreatePowerRequest	The PoCreatePowerRequest routine creates a power request object.
PoDeletePowerRequest	The PoDeletePowerRequest routine deletes a power request object.
PoEndDeviceBusy	The PoEndDeviceBusy routine marks the end of a period of time in which the device is busy.
PoQueryWatchdogTime	The PoQueryWatchdogTime routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.
PoRegisterDeviceForIdleDetection	The PoRegisterDeviceForIdleDetection routine enables or cancels idle detection and sets idle time-out values for a device.
PoRegisterPowerSettingCallback	The PoRegisterPowerSettingCallback routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.

TITLE	DESCRIPTION
PoRegisterSystemState	The PoRegisterSystemState routine registers the system as busy due to certain activity.
PoSetDeviceBusyEx	The PoSetDeviceBusyEx routine notifies the power manager that the device associated with the specified idle counter is busy.
PoSetPowerRequest	The PoSetPowerRequest routine increments the count for the specified power request type.
PoSetPowerState	The PoSetPowerState routine notifies the system of a change in the device power state for a device.
PoStartDeviceBusy	The PoStartDeviceBusy routine marks the start of a period of time in which the device is busy.
PoStartNextPowerIrp	The PoStartNextPowerIrp routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoUnregisterPowerSettingCallback	The PoUnregisterPowerSettingCallback routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.
PoUnregisterSystemState	The PoUnregisterSystemState routine cancels a system state registration created by PoRegisterSystemState.
PsChargePoolQuota	The PsChargePoolQuota routine charges pool quota of the specified pool type to the specified process.
PsDereferenceImpersonationToken	The PsDereferenceImpersonationToken routine decrements the reference count of an impersonation token.
PsDereferencePrimaryToken	The PsDereferencePrimaryToken routine decrements the reference count of a primary token.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetProcessExitTime	The PsGetProcessExitTime routine returns the exit time for the current process.
PsImpersonateClient	The PsImpersonateClient routine causes a server thread to impersonate a client.
PsIsDiskCountersEnabled	The enabled state of the per process disk I/O counters is returned by the PsIsDiskCountersEnabled routine.
PsIsSystemThread	The PsIsSystemThread routine checks whether a given thread is a system thread.
PsIsThreadTerminating	The PsIsThreadTerminating routine checks whether a thread is terminating.

TITLE	DESCRIPTION
PsLookupProcessByProcessId	The PsLookupProcessByProcessId routine accepts the process ID of a process and returns a referenced pointer to EPROCESS structure of the process.
PsLookupThreadByThreadId	The PsLookupThreadByThreadId routine accepts the thread ID of a thread and returns a referenced pointer to the ETHREAD structure of the thread.
PsReferenceImpersonationToken	The PsReferenceImpersonationToken routine increments the reference count of the impersonation token for the specified thread.
PsReferencePrimaryToken	The PsReferencePrimaryToken routine increments the reference count of the primary token for the specified process.
PsReturnPoolQuota	The PsReturnPoolQuota routine returns pool quota of the specified pool type to the specified process.
PsRevertToSelf	The PsRevertToSelf routine ends the calling thread's impersonation of a client.
PsUpdateDiskCounters	The PsUpdateDiskCounters routine updates the disk I/O counters of a given process.
RtlAbsoluteToSelfRelativeSD	The RtlAbsoluteToSelfRelativeSD routine creates a new security descriptor in self-relative format by using a security descriptor in absolute format as a template.
RtlAddAccessAllowedAce	The RtlAddAccessAllowedAce routine adds an access-allowed access control entry (ACE) to an access control list (ACL). The access is granted to the specified security identifier (SID).
RtlAddAccessAllowedAceEx	The RtlAddAccessAllowedAceEx routine adds an access-allowed access control entry (ACE) with inheritance ACE flags to an access control list (ACL). The access is granted to the specified security identifier (SID).
RtlAddAce	The RtlAddAce routine adds one or more access control entries (ACEs) to a specified access control list (ACL).
RtlAllocateAndInitializeSid	Reserved for system use.
RtlAllocateHeap	The RtlAllocateHeap routine allocates a block of memory from a heap.
RtlAppendStringToString	The RtlAppendStringToString routine concatenates two counted strings. It copies bytes from the source up to the length of the destination buffer.
RtlCaptureContext	The RtlCaptureContext function retrieves a context record in the context of the caller.
RtlCaptureStackBackTrace	The RtlCaptureStackBackTrace routine captures a stack back trace by walking up the stack and recording the information for each frame.

TITLE	DESCRIPTION
RtlCompareMemoryUlong	The RtlCompareMemoryUlong routine returns how many bytes in a block of memory match a specified pattern.
RtlCompressBuffer	The RtlCompressBuffer function compresses a buffer and can be used by a file system driver to facilitate the implementation of file compression.
RtlCompressChunks	Reserved for system use.
RtlConvertSidToUnicodeString	The RtlConvertSidToUnicodeString routine generates a printable Unicode string representation of a security identifier (SID).
RtlCopyLuid	The RtlCopyLuid routine copies a locally unique identifier (LUID) to a buffer.
RtlCopySid	The RtlCopySid routine copies the value of a security identifier (SID) to a buffer.
RtlCreateAcl	The RtlCreateAcl routine creates and initializes an access control list (ACL).
RtlCreateHeap	The RtlCreateHeap routine creates a heap object that can be used by the calling process. This routine reserves space in the virtual address space of the process and allocates physical storage for a specified initial portion of this block.
RtlCreateSecurityDescriptorRelative	The RtlCreateSecurityDescriptorRelative routine initializes a new security descriptor in self-relative format.
RtlCreateSystemVolumeInformationFolder	The RtlCreateSystemVolumeInformationFolder routine verifies the existence of the "System Volume Information" folder on a file system volume. If the folder is not present, then the folder is created.
RtlCreateUnicodeString	The RtlCreateUnicodeString routine creates a new counted Unicode string.
RtlCustomCPToUnicodeN	Reserved for system use.
RtlDecompressBuffer	The RtlDecompressBuffer function decompresses an entire compressed buffer.
RtlDecompressBufferEx	The RtlDecompressBufferEx function decompresses an entire compressed buffer.
RtlDecompressBufferEx2	The RtlDecompressBufferEx2 function decompresses an entire compressed buffer, using multiple processors where possible. Multiple processor support is only implemented for kernel mode callers.

TITLE	DESCRIPTION
RtlDecompressChunks	Reserved for system use.
RtlDecompressFragment	The RtlDecompressFragment function is used to decompress part of a compressed buffer (that is, a buffer "fragment").
RtlDecompressFragmentEx	The RtlDecompressFragmentEx function is used to decompress part of a compressed buffer (that is, a buffer "fragment"), using multiple processors where possible.
RtlDeleteAce	The RtlDeleteAce routine deletes an access control entry (ACE) from a specified access control list (ACL).
RtlDescribeChunk	Reserved for system use.
RtlDestroyHeap	The RtlDestroyHeap routine destroys the specified heap object. RtlDestroyHeap decommits and releases all the pages of a private heap object, and it invalidates the handle to the heap.
RtlDowncaseUnicodeString	The RtlDowncaseUnicodeString routine converts the specified Unicode source string to lowercase. The translation conforms to the current system locale information.
RtlEqualPrefixSid	The RtlEqualPrefixSid routine determines whether two security-identifier (SID) prefixes are equal. An SID prefix is the entire SID except for the last subauthority value.
RtlEqualSid	The RtlEqualSid routine determines whether two security identifier (SID) values are equal. Two SIDs must match exactly to be considered equal.
RtlFillMemoryUlong	The RtlFillMemoryUlong routine fills the specified range of memory with one or more repetitions of a ULONG value.
RtlFillMemoryUlonglong	The RtlFillMemoryUlonglong routine fills a given range of memory with one or more repetitions of a given ULONGLONG value.
RtlFindUnicodePrefix	The RtlFindUnicodePrefix routine searches for the best match for a given Unicode file name in a prefix table.
RtlFreeHeap	The RtlFreeHeap routine frees a memory block that was allocated from a heap by RtlAllocateHeap.
RtlFreeOemString	The RtlFreeOemString routine releases storage that was allocated by any of the Rtl..ToOemString routines.
RtlFreeSid	Reserved for system use.
RtlGenerate8dot3Name	The RtlGenerate8dot3Name routine generates a short (8.3) name for the specified long file name.
RtlGetAce	The RtlGetAce routine obtains a pointer to an access control entry (ACE) in an access control list (ACL).

TITLE	DESCRIPTION
RtlGetCompressionWorkSpaceSize	The RtlGetCompressionWorkSpaceSize function is used to determine the correct size of the WorkSpace buffer for the RtlCompressBuffer and RtlDecompressFragment functions.
RtlGetDaclSecurityDescriptor	The RtlGetDaclSecurityDescriptor routine returns a pointer to the discretionary ACL (DACL) for a security descriptor.
RtlGetGroupSecurityDescriptor	The RtlGetGroupSecurityDescriptor routine returns the primary group information for a given security descriptor.
RtlGetOwnerSecurityDescriptor	The RtlGetOwnerSecurityDescriptor routine returns the owner information for a given security descriptor.
RtlGetSaclSecurityDescriptor	The RtlGetSaclSecurityDescriptor routine returns a pointer to the system ACL (SACL) for a security descriptor.
RtlIdentifierAuthoritySid	Reserved for system use.
RtlInitCodePageTable	Reserved for system use.
RtlInitializeSid	The RtlInitializeSid routine initializes a security identifier (SID) structure.
RtlInitializeSidEx	The RtlInitializeSidEx routine initializes a pre-allocated security identifier (SID) structure.
RtlInitializeUnicodePrefix	The RtlInitializeUnicodePrefix routine initializes a prefix table.
RtlInitStringEx	The RtlInitStringEx routine initializes a counted string of 8-bit characters.
RtlInsertUnicodePrefix	The RtlInsertUnicodePrefix routine inserts a new element into a Unicode prefix table.
RtlIsCloudFilesPlaceholder	The RtlIsCloudFilesPlaceholder routine determines if a file or a directory is a CloudFiles placeholder, based on the FileAttributes and ReparseTag values of the file.
RtlIsNameLegalDOS8Dot3	The RtlIsNameLegalDOS8Dot3 routine determines whether a given name represents a valid short (8.3) file name.
RtlIsPartialPlaceholder	The RtlIsPartialPlaceholder routine determines if a file or a directory is a CloudFiles placeholder, based on the FileAttributes and ReparseTag values of the file.
RtlIsPartialPlaceholderFileHandle	The RtlIsPartialPlaceholderFileHandle routine determines if a file is a known type of placeholder, based on a file handle.
RtlIsPartialPlaceholderFileInfo	The RtlIsPartialPlaceholderFileInfo routine determines if a file is a known type of placeholder, based on the information returned by NtQueryInformationFile or NtQueryDirectoryFile.
RtlIsValidOemCharacter	The RtlIsValidOemCharacter routine determines if the specified Unicode character can be mapped to a valid OEM character.

TITLE	DESCRIPTION
RtlLengthRequiredSid	The RtlLengthRequiredSid routine returns the length, in bytes, of the buffer required to store a security identifier (SID) with a specified number of subauthorities.
RtlLengthSid	The RtlLengthSid routine returns the length, in bytes, of a valid security identifier (SID).
RtlMultiByteToUnicodeN	The RtlMultiByteToUnicodeN routine translates the specified source string into a Unicode string, using the current system ANSI code page (ACP). The source string is not necessarily from a multibyte character set.
RtlMultiByteToUnicodeSize	The RtlMultiByteToUnicodeSize routine determines the number of bytes that are required to store the Unicode translation for the specified source string.
RtlNextUnicodePrefix	The RtlNextUnicodePrefix routine is used to enumerate the elements in a Unicode prefix table.
RtlNtStatusToDosError	The RtlNtStatusToDosError routine converts the specified NTSTATUS code to its equivalent system error code.
RtlNtStatusToDosErrorNoTeb	Reserved for system use.
RtlOemStringToCountedUnicodeSize	The RtlOemStringToCountedUnicodeSize routine determines the size, in bytes, that a given OEM string will be after it is translated into a counted Unicode string.
RtlOemStringToCountedUnicodeString	The RtlOemStringToCountedUnicodeString routine translates the specified source string into a Unicode string using the current system OEM code page.
RtlOemStringToUnicodeSize	The RtlOemStringToUnicodeSize routine determines the size, in bytes, that a given OEM string will be after it is translated into a null-terminated Unicode string.
RtlOemStringToUnicodeString	The RtlOemStringToUnicodeString routine translates a given source string into a null-terminated Unicode string using the current system OEM code page.
RtlOemToUnicodeN	The RtlOemToUnicodeN routine translates the specified source string into a Unicode string, using the current system OEM code page.
RtlQueryProcessPlaceholderCompatibilityMode	The RtlSetProcessPlaceholderCompatibilityMode function returns the placeholder compatibility mode for the current process.
RtlQueryThreadPlaceholderCompatibilityMode	RtlQueryThreadPlaceholderCompatibilityMode is a routine which returns the placeholder compatibility mode for the current thread.
RtlRandom	The RtlRandom routine returns a random number that was generated from a given seed value.

TITLE	DESCRIPTION
RtlRandomEx	The RtlRandomEx routine returns a random number that was generated from a given seed value.
RtlRemoveUnicodePrefix	The RtlRemoveUnicodePrefix routine removes an element from a prefix table.
RtlReserveChunk	Reserved for system use.
RtlSecondsSince1970ToTime	The RtlSecondsSince1970ToTime routine converts the elapsed time, in seconds, since the beginning of 1970 to an absolute system time value.
RtlSecondsSince1980ToTime	The RtlSecondsSince1980ToTime routine converts the elapsed time, in seconds, since the beginning of 1980 to an absolute system time value.
RtlSelfRelativeToAbsoluteSD	The RtlSelfRelativeToAbsoluteSD routine creates a new security descriptor in absolute format by using a security descriptor in self-relative format as a template.
RtlSetGroupSecurityDescriptor	The RtlSetGroupSecurityDescriptor routine sets the primary group information of an absolute-format security descriptor. It replaces any primary group information that is already present in the security descriptor.
RtlSetOwnerSecurityDescriptor	The RtlSetOwnerSecurityDescriptor routine sets the owner information of an absolute-format security descriptor. It replaces any owner information that is already present in the security descriptor.
RtlSetProcessPlaceholderCompatibilityMode	The RtlSetProcessPlaceholderCompatibilityMode function returns the placeholder compatibility mode for the current process.
RtlSetThreadPlaceholderCompatibilityMode	RtlSetThreadPlaceholderCompatibilityMode is a routine which sets the placeholder compatibility mode for the current thread.
RtlSubAuthorityCountSid	Reserved for system use.
RtlSubAuthoritySid	The RtlSubAuthoritySid routine returns a pointer to a specified subauthority of a security identifier (SID).
RtlTimeToSecondsSince1970	The RtlTimeToSecondsSince1970 routine converts a given absolute system time value to the elapsed time, in seconds, since the beginning of 1970.
RtlTimeToSecondsSince1980	The RtlTimeToSecondsSince1980 routine converts a given absolute system time value to the elapsed time, in seconds, since the beginning of 1980.
RtlUnicodeStringToCountedOemString	The RtlUnicodeStringToCountedOemString routine translates the specified Unicode source string into a counted OEM string using the current system OEM code page.

TITLE	DESCRIPTION
RtlUnicodeStringToOemSize	The RtlUnicodeStringToOemSize routine determines the size, in bytes, that a given Unicode string will be after it is translated into an OEM string.
RtlUnicodeStringToOemString	The RtlUnicodeStringToOemString routine translates a given Unicode source string into an OEM string using the current system OEM code page.
RtlUnicodeToCustomCPN	Reserved for system use.
RtlUnicodeToMultiByteN	The RtlUnicodeToMultiByteN routine translates the specified Unicode string into a new character string, using the current system ANSI code page (ACP). The translated string is not necessarily from a multibyte character set.
RtlUnicodeToMultiByteSize	The RtlUnicodeToMultiByteSize routine determines the number of bytes that are required to store the multibyte translation for the specified Unicode string. The translation is assumed to use the current system ANSI code page (ACP).
RtlUnicodeToOemN	The RtlUnicodeToOemN routine translates a given Unicode string to an OEM string, using the current system OEM code page.
RtlUnicodeToUTF8N	The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.
RtlUpcaseUnicodeStringToCountedOemString	The RtlUpcaseUnicodeStringToCountedOemString routine translates a given Unicode source string into an uppercase counted OEM string using the current system OEM code page.
RtlUpcaseUnicodeStringToOemString	The RtlUpcaseUnicodeStringToOemString routine translates a given Unicode source string into an uppercase OEM string using the current system OEM code page.
RtlUpcaseUnicodeToCustomCPN	Reserved for system use.
RtlUpcaseUnicodeToMultiByteN	The RtlUpcaseUnicodeToMultiByteN routine translates the specified Unicode string into a new uppercase character string, using the current system ANSI code page (ACP). The translated string is not necessarily from a multibyte character set.
RtlUpcaseUnicodeToOemN	The RtlUpcaseUnicodeToOemN routine translates a given Unicode string into an uppercase OEM string, using the current system OEM code page.
RtlUTF8ToUnicodeN	The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.
RtlValidSid	The RtlValidSid routine validates a security identifier (SID) by verifying that the revision number is within a known range and that the number of subauthorities is less than the maximum.

TITLE	DESCRIPTION
RtlxOemStringToUnicodeSize	Reserved for system use.
RtlxUnicodeStringToOemSize	Reserved for system use.
SeAppendPrivileges	The SeAppendPrivileges routine appends additional privileges to the privilege set in an access state structure.
SeAuditHardLinkCreation	Reserved for system use.
SeAuditingFileEvents	The SeAuditingFileEvents routine determines whether file open events are currently being audited.
SeAuditingFileOrGlobalEvents	The SeAuditingFileOrGlobalEvents routine determines whether file or global events are currently being audited.
SeAuditingHardLinkEvents	Reserved for system use.
SeCaptureSubjectContext	The SeCaptureSubjectContext routine captures the security context of the calling thread for access validation and auditing.
SeCreateClientSecurity	The SeCreateClientSecurity routine initializes a security client context structure with the information needed to call SeImpersonateClientEx.
SeCreateClientSecurityFromSubjectContext	The SeCreateClientSecurityFromSubjectContext routine retrieves the access token for a security subject context and uses the result to initialize a security client context with the information needed to call SeImpersonateClientEx.
SeDeleteClientSecurity	The SeDeleteClientSecurity routine deletes a client security context.
SeDeleteObjectAuditAlarm	The SeDeleteObjectAuditAlarm routine generates audit and alarm messages for an object that is marked for deletion.
SeFilterToken	The SeFilterToken routine creates a new access token that is a restricted version of an existing access token.
SeFreePrivileges	The SeFreePrivileges routine frees a privilege set returned by SeAccessCheck.
SeImpersonateClient	Obsolete.
SeImpersonateClientEx	The SeImpersonateClientEx routine causes a thread to impersonate a user.
SeLengthSid	Obsolete.
SeLockSubjectContext	The SeLockSubjectContext routine locks the primary and impersonation tokens of a captured subject context.

TITLE	DESCRIPTION
SeMarkLogonSessionForTerminationNotification	The SeMarkLogonSessionForTerminationNotification routine marks a logon session so that the caller's registered callback routine is called when the logon session terminates.
SeOpenObjectAuditAlarm	The SeOpenObjectAuditAlarm routine generates audit and alarm messages when an attempt is made to open an object.
SeOpenObjectForDeleteAuditAlarm	The SeOpenObjectForDeleteAuditAlarm routine generates audit and alarm messages when an attempt is made to open an object for deletion.
SePrivilegeCheck	The SePrivilegeCheck routine determines whether a specified set of privileges is enabled in the subject's access token.
SeQueryAuthenticationIdToken	The SeQueryAuthenticationIdToken routine retrieves the authentication ID of an access token.
SeQueryInformationToken	The SeQueryInformationToken routine retrieves a specified type of information about an access token. The calling process must have appropriate access rights to obtain the information.
SeQuerySecurityDescriptorInfo	The SeQuerySecurityDescriptorInfo routine retrieves a copy of an object's security descriptor.
SeQuerySessionIdToken	Reserved for system use.
SeQuerySubjectContextToken	The SeQuerySubjectContextToken macro retrieves the access token for a security subject context.
SeRegisterLogonSessionTerminatedRoutine	The SeRegisterLogonSessionTerminatedRoutine routine registers a callback routine to be called when a logon session terminates. A logon session terminates when the last token referencing the logon session is deleted.
SeReleaseSubjectContext	The SeReleaseSubjectContext routine releases a subject security context captured by an earlier call to SeCaptureSubjectContext.
SeSetAccessStateGenericMapping	The SeSetAccessStateGenericMapping routine sets the generic mapping field of an ACCESS_STATE structure.
SeSetSecurityDescriptorInfo	The SeSetSecurityDescriptorInfo routine sets an object's security descriptor.
SeSetSecurityDescriptorInfoEx	The SeSetSecurityDescriptorInfoEx routine modifies an object's security descriptor and specifies whether the object supports automatic inheritance of access control entries (ACE).
SeSetSessionIdToken	Reserved for system use.
SeTokenGetNoChildProcessRestricted	The SeTokenGetNoChildProcessRestricted routine determines the state of the no child process mitigation. It is not possible to be enforced and audit-only at the same time.

TITLE	DESCRIPTION
SeTokenIsAdmin	The SeTokenIsAdmin routine determines whether a token contains the local administrators group.
SeTokenIsNoChildProcessRestrictionEnforced	The SeTokenIsNoChildProcessRestrictionEnforced routine determines if the token carries the no child process restriction.
SeTokenIsRestricted	The SeTokenIsRestricted routine determines whether a token contains a list of restricting security identifiers (SID).
SeTokenSetNoChildProcessRestricted	The SeTokenSetNoChildProcessRestricted routine sets the TOKEN_AUDIT_NO_CHILD_PROCESS or TOKEN_AUDIT_NO_CHILD_PROCESS flags in the token.
SeTokenType	Reserved for system use.
SeUnlockSubjectContext	The SeUnlockSubjectContext routine unlocks the tokens of a captured subject context that were locked by a call to SeLockSubjectContext.
SeUnregisterLogonSessionTerminatedRoutine	The SeUnregisterLogonSessionTerminatedRoutine routine unregisters a callback routine that was registered by an earlier call to SeRegisterLogonSessionTerminatedRoutine.

Callback functions

TITLE	DESCRIPTION
DRIVER_FS_NOTIFICATION	A PDRIVER_FS_NOTIFICATION-typed routine is called by the operating system when a file system registers or unregisters itself by using IoRegisterFileSystem or IoUnregisterFileSystem.

Structures

TITLE	DESCRIPTION
ACCESS_ALLOWED_ACE	The ACCESS_ALLOWED_ACE structure defines an access-control entry (ACE) for the discretionary access-control list (DACL) that controls access to an object.
ACCESS_DENIED_ACE	The ACCESS_DENIED_ACE structure defines an access-control entry (ACE) for the discretionary access-control list (DACL) controlling access to an object.
ACE_HEADER	The ACE_HEADER structure describes the type and size of an access-control entry (ACE).
BOOT_AREA_INFO	The BOOT_AREA_INFO structure contains the output for the FSCTL_GET_BOOT_AREA_INFO control code.
ENCRYPTION_KEY_CTRL_INPUT	

TITLE	DESCRIPTION
FILE_ACCESS_INFORMATION	The FILE_ACCESS_INFORMATION structure is used to query for or set the access rights of a file.
FILE_ALL_INFORMATION	The FILE_ALL_INFORMATION structure is a container for several FILE_XXX_INFORMATION structures.
FILE_ALLOCATION_INFORMATION	The FILE_ALLOCATION_INFORMATION structure is used to set the allocation size for a file.
FILE_BOTH_DIR_INFORMATION	The FILE_BOTH_DIR_INFORMATION structure is used to query detailed information for the files in a directory.
FILE_CASE_SENSITIVE_INFORMATION	The FILE_CASE_SENSITIVE_INFORMATION structure is used to query or set per-directory case-sensitive information.
FILE_COMPLETION_INFORMATION	The FILE_COMPLETION_INFORMATION structure contains the port handle and key for an I/O completion port created for a file handle.
FILE_COMPRESSION_INFORMATION	The FILE_COMPRESSION_INFORMATION structure describes the state of a compressed data buffer.
FILE_DESIRED_STORAGE_CLASS_INFORMATION	Contains the information for the Desired Storage Class attribute.
FILE_DIRECTORY_INFORMATION	The FILE_DIRECTORY_INFORMATION structure is used to query detailed information for the files in a directory.
FILE_EA_INFORMATION	The FILE_EA_INFORMATION structure is used to query for the size of the extended attributes (EA) for a file.
FILE_FS_ATTRIBUTE_INFORMATION	The FILE_FS_ATTRIBUTE_INFORMATION structure is used to query attribute information for a file system.
FILE_FS_CONTROL_INFORMATION	The FILE_FS_CONTROL_INFORMATION structure is used to query or set control information for the files in a directory.
FILE_FS_DRIVER_PATH_INFORMATION	The FILE_FS_DRIVER_PATH_INFORMATION structure is used to query whether a given driver is in the I/O path for a file system volume.
FILE_FS_PERSISTENT_VOLUME_INFORMATION	The FILE_FS_PERSISTENT_VOLUME_INFORMATION structure is used to control persistent settings for a file system volume. Persistent settings persist on a file system volume between reboots of the computer.
FILE_FULL_DIR_INFORMATION	The FILE_FULL_DIR_INFORMATION structure is used to query detailed information for the files in a directory.
FILE_GET_EA_INFORMATION	The FILE_GET_EA_INFORMATION structure is used to query for extended-attribute (EA) information.
FILE_GET_QUOTA_INFORMATION	The FILE_GET_QUOTA_INFORMATION structure is used to query for quota information.

TITLE	DESCRIPTION
FILE_ID_BOTH_DIR_INFORMATION	The FILE_ID_BOTH_DIR_INFORMATION structure is used to query file reference number information for the files in a directory.
FILE_ID_EXTD_BOTH_DIR_INFORMATION	The FILE_ID_EXTD_BOTH_DIR_INFORMATION structure is used to query file reference number information for the files in a directory.
FILE_ID_FULL_DIR_INFORMATION	The FILE_ID_FULL_DIR_INFORMATION structure is used to query detailed information for the files in a directory.
FILE_ID_GLOBAL_TX_DIR_INFORMATION	The FILE_ID_GLOBAL_TX_DIR_INFORMATION structure contains information about transactional visibility for the files in a directory.
FILE_INTERNAL_INFORMATION	The FILE_INTERNAL_INFORMATION structure is used to query for the file system's 8-byte file reference number for a file.
FILE_LEVEL_TRIM	The FILE_LEVEL_TRIM structure contains an array of byte ranges to trim for a file.
FILE_LEVEL_TRIM_OUTPUT	The FILE_LEVEL_TRIM_OUTPUT structure contains the results of a trim operation performed by an FSCTL_FILE_LEVEL_TRIM request.
FILE_LEVEL_TRIM_RANGE	Contains the offset and length of a trim range for a file.
FILE_LINK_ENTRY_INFORMATION	The FILE_LINK_ENTRY_INFORMATION structure describes a single NTFS hard link to an existing file.
FILE_LINK_INFORMATION	The FILE_LINK_INFORMATION structure is used to create an NTFS hard link to an existing file.
FILE_LINKS_INFORMATION	The FILE_LINKS_INFORMATION structure is used to query NTFS hard links to an existing file.
FILE_MAILSLOT_QUERY_INFORMATION	The FILE_MAILSLOT_QUERY_INFORMATION structure contains information about a mailslot.
FILE_MAILSLOT_SET_INFORMATION	The FILE_MAILSLOT_SET_INFORMATION structure is used to set a value on a mailslot.
FILE_MODE_INFORMATION	The FILE_MODE_INFORMATION structure is used to query or set the access mode of a file.
FILE_NAMES_INFORMATION	A FILE_NAMES_INFORMATION structure used to query detailed information about the names of files in a directory.
FILE_NETWORK_PHYSICAL_NAME_INFORMATION	Contains the full UNC physical pathname for a file or directory on a remote file share.
FILE_OBJECTID_INFORMATION	The FILE_OBJECTID_INFORMATION structure is used to query for object ID information for the files in a directory on an NTFS volume.

TITLE	DESCRIPTION
FILE_PIPE_INFORMATION	The FILE_PIPE_INFORMATION structure contains information about a named pipe that is not specific to the local or the remote end of the pipe.
FILE_PIPE_LOCAL_INFORMATION	The FILE_PIPE_LOCAL_INFORMATION structure contains information about the local end of a named pipe.
FILE_PIPE_REMOTE_INFORMATION	The FILE_PIPE_REMOTE_INFORMATION structure contains information about the remote end of a named pipe.
FILE_PROVIDER_EXTERNAL_INFO_V0	This structure may be altered or unavailable. Instead, use FILE_PROVIDER_EXTERNAL_INFO_V1.
FILE_PROVIDER_EXTERNAL_INFO_V1	The FILE_PROVIDER_EXTERNAL_INFO_V1 structure defines metadata specific to files provided by WOF_PROVIDER_FILE.
FILE_QUOTA_INFORMATION	The FILE_QUOTA_INFORMATION structure is used to query or set per-user quota information for each of the files in a directory.
FILE_RENAME_INFORMATION	The FILE_RENAME_INFORMATION structure is used to rename a file.
FILE_REPARSE_POINT_INFORMATION	The FILE_REPARSE_POINT_INFORMATION structure is used to query for information about a reparse point.
FILE_STAT_INFORMATION	Contains metadata about a file.
FILE_STAT_LX_INFORMATION	The _FILE_STAT_LX_INFORMATION structure contains metadata about a file.
FILE_STORAGE_RESERVE_ID_INFORMATION	Queries and sets the storage reserve ID on a file or directory.
FILE_STREAM_INFORMATION	The FILE_STREAM_INFORMATION structure is used to enumerate the streams for a file.
FILE_ZERO_DATA_INFORMATION	Contains a range of a file to set to zeros.
FILE_ZERO_DATA_INFORMATION_EX	Contains a range of a file to set to zeros.
FS_FILTER_SECTION_SYNC_OUTPUT	The FS_FILTER_SECTION_SYNC_OUTPUT structure contains information describing the attributes of the section that is being created.
FSCTL_OFFLOAD_READ_INPUT	The FSCTL_OFFLOAD_READ_INPUT structure contains the input for the FSCTL_OFFLOAD_READ control code request.
FSCTL_OFFLOAD_READ_OUTPUT	The FSCTL_OFFLOAD_READ_OUTPUT structure contains the output for the FSCTL_OFFLOAD_READ control code request.
FSCTL_OFFLOAD_WRITE_INPUT	The FSCTL_OFFLOAD_WRITE_INPUT structure contains the input for the FSCTL_OFFLOAD_WRITE control code request.

TITLE	DESCRIPTION
FSCTL_OFFLOAD_WRITE_OUTPUT	The FSCTL_OFFLOAD_WRITE_OUTPUT structure contains the output for the FSCTL_OFFLOAD_WRITE control code request.
FSCTL_QUERY_VOLUME_NUMA_INFO_OUTPUT	The FSCTL_QUERY_VOLUME_NUMA_INFO_OUTPUT structure specifies the Non-Uniform Memory Architecture (NUMA) node the volume resides on.
FSRTL_ADVANCED_FCB_HEADER	The FSRTL_ADVANCED_FCB_HEADER structure contains context information that a file system maintains about a file.
FSRTL_COMMON_FCB_HEADER	Do not use the FSRTL_COMMON_FCB_HEADER structure outside of the FSRTL_ADVANCED_FCB_HEADER structure.
IO_PRIORITY_INFO	The IO_PRIORITY_INFO structure is used to hold thread priority information.
MEMORY_BASIC_INFORMATION	Contains information about a range of pages in the virtual address space of a process.
NETWORK_APP_INSTANCE_EA	
PUBLIC_OBJECT_BASIC_INFORMATION	The PUBLIC_OBJECT_BASIC_INFORMATION structure holds a subset of the full information that is available for an object.
PUBLIC_OBJECT_TYPE_INFORMATION	The PUBLIC_OBJECT_TYPE_INFORMATION structure holds the type name of the object.
QUERY_FILE_LAYOUT_INPUT	The QUERY_FILE_LAYOUT_INPUT structure selects which file layout entries are returned from a FSCTL_QUERY_FILE_LAYOUT request.
QUERY_FILE_LAYOUT_OUTPUT	The QUERY_FILE_LAYOUT_OUTPUT structure serves as a header for the file layout entries that are returned from a FSCTL_QUERY_FILE_LAYOUT request.
REFS_SMR_VOLUME_GC_PARAMETERS	The REFS_SMR_VOLUME_GC_PARAMETERS structure.
REFS_SMR_VOLUME_INFO_OUTPUT	The REFS_SMR_VOLUME_INFO_OUTPUT structure describes a Shingled Magnetic Recording (SMR) volume's current state on space and garbage collection activities.
REPARSE_DATA_BUFFER	The REPARSE_DATA_BUFFER structure contains reparse point data for a Microsoft reparse point.
REPARSE_DATA_BUFFER_EX	
REPARSE_GUID_DATA_BUFFER	The REPARSE_GUID_DATA_BUFFER structure contains reparse point data for a reparse point.
RTL_SEGMENT_HEAP_MEMORY_SOURCE	The RTL_SEGMENT_HEAP_MEMORY_SOURCE structure specifies the segment heap memory source.
RTL_SEGMENT_HEAP_PARAMETERS	The RTL_SEGMENT_HEAP_PARAMETERS structure contains the segment heap parameters.

TITLE	DESCRIPTION
SE_EXPORTS	The SeExports structure is a large external static SE_EXPORTS structure that defines a number of well-known security constants for privilege values and security identifiers.
SE_SID	The SE_SID union holds the maximum-sized valid Security Identifier (SID). The structure occupies 68-bytes and is suitable for stack allocation.
SE_TOKEN_USER	The SE_TOKEN_USER structure holds the maximum-sized valid user SID that can be returned by SeQueryInformationToken, GetTokenInformation, or ZwQueryInformationToken with the TokenUser information class. This structure is suitable for stack allocation.
SECURITY_DESCRIPTOR	The SECURITY_DESCRIPTOR structure contains the security information associated with an object. Drivers use this structure to set and query an object's security status.
SET_DAX_ALLOC_ALIGNMENT_HINT_INPUT	This structure is for internal use only and should not be called from your code.
SID	The security identifier (SID) structure is a variable-length structure used to uniquely identify users or groups.
SID_AND_ATTRIBUTES	The SID_AND_ATTRIBUTES structure represents a security identifier (SID) and its attributes. SIDs are used to uniquely identify users or groups.
SID_IDENTIFIER_AUTHORITY	The SID_IDENTIFIER_AUTHORITY structure represents the top-level authority of a security identifier (SID).
SYSTEM_ALARM_ACE	Reserved for future use.
SYSTEM_AUDIT_ACE	The SYSTEM_AUDIT_ACE structure defines an access-control entry (ACE) for the system access-control list (ACL) specifying what types of access cause system-level notifications.
SYSTEM_PROCESS_TRUST_LABEL_ACE	Reserved.
SYSTEM_RESOURCE_ATTRIBUTE_ACE	The SYSTEM_RESOURCE_ATTRIBUTE_ACE structure defines an access-control entry (ACE) for the system access-control list (ACL) specifying what rights a particular claim has to a resource.
SYSTEM_SCOPED_POLICY_ID_ACE	The SYSTEM_SCOPED_POLICY_ID_ACE structure defines an access-control entry (ACE) for the system access-control list (ACL) specifying rights for a scoped policy identifer.
TOKEN_CONTROL	The TOKEN_CONTROL structure contains information that identifies an access token.
TOKEN_DEFAULT_DACL	The TOKEN_DEFAULT_DACL structure specifies a discretionary access-control list (DACL).

TITLE	DESCRIPTION
TOKEN_GROUPS	TOKEN_GROUPS contains information about the group security identifiers (SID) in an access token.
TOKEN_GROUPS_AND_PRIVILEGES	TOKEN_GROUPS_AND_PRIVILEGES contains information about the group security identifiers (SIDs) and privileges in an access token.
TOKEN_ORIGIN	The TOKEN_ORIGIN structure contains information about the origin of the logon session.
TOKEN_OWNER	TOKEN_OWNER contains the default owner security identifier (SID) that will be applied to newly created objects.
TOKEN_PRIMARY_GROUP	TOKEN_PRIMARY_GROUP specifies a group security identifier (SID) for an access token.
TOKEN_PRIVILEGES	TOKEN_PRIVILEGES contains information about a set of privileges for an access token.
TOKEN_SOURCE	TOKEN_SOURCE identifies the source of an access token.
TOKEN_STATISTICS	TOKEN_STATISTICS contains information about an access token. A driver can retrieve this information by calling SeQueryInformationToken or ZwQueryInformationToken.
TOKEN_USER	TOKEN_USER identifies the user associated with an access token.
VIRTUAL_STORAGE_SET_BEHAVIOR_INPUT	
VIRTUALIZATION_INSTANCE_INFO_INPUT_EX	
WIM_PROVIDER_ADD_OVERLAY_INPUT	A new Windows Image File (WIM) data source is added to the WIM provider with the WIM_PROVIDER_ADD_OVERLAY_INPUT structure.
WIM_PROVIDER_EXTERNAL_INFO	The WIM_PROVIDER_EXTERNAL_INFO structure holds the identifier and status information for the Windows Image File (WIM) external backing provider.
WIM_PROVIDER_OVERLAY_ENTRY	Contains the a Windows Image Format (WIM) file configuration information for a data source entry. It is used to identify specific WIM file names and indices that supply data to externally backed files on a volume.
WIM_PROVIDER_REMOVE_OVERLAY_INPUT	A Windows Image File (WIM) data source to remove from the WIM provider is specified in the WIM_PROVIDER_REMOVE_OVERLAY_INPUT structure.
WIM_PROVIDER_SUSPEND_OVERLAY_INPUT	A Windows Image File (WIM) data source to suspend from the WIM provider is specified in the WIM_PROVIDER_SUSPEND_OVERLAY_INPUT structure.

TITLE	DESCRIPTION
WIM_PROVIDER_UPDATE_OVERLAY_INPUT	A current Windows Image File (WIM) data source is updated with a new WIM file using the FSCTL_UPDATE_OVERLAY control request with a WIM_PROVIDER_UPDATE_OVERLAY_INPUT structure.
WOF_EXTERNAL_FILE_ID	The WOF_EXTERNAL_FILE_ID structure contains a file ID that is used to open a handle to a mini-filter or driver.
WOF_EXTERNAL_INFO	The WOF_EXTERNAL_INFO structure identifies a file backing provider and the overlay service version it supports.
WOF_VERSION_INFO	The WOF_VERSION_INFO structure contains the version corresponding to the driver supporting a given provider.

Enumerations

TITLE	DESCRIPTION
CSV_CONTROL_OP	Specifies the type of CSV control operation to use with the FSCTL_CSV_CONTROL control code.
FILE_STORAGE_TIER_CLASS	Defines values for the type of desired storage class.
MEMORY_INFORMATION_CLASS	Defines classes of memory information that can be retrieved by using the ZwQueryVirtualMemory function.
OBJECT_INFORMATION_CLASS	The OBJECT_INFORMATION_CLASS enumeration type represents the type of information to supply about an object.
REFS_SMR_VOLUME_GC_ACTION	The REFS_SMR_VOLUME_GC_ACTION enum contains the available garbage collection commands for FSCTL_SET_REFS_SMR_VOLUME_GC_PARAMETERS.
REFS_SMR_VOLUME_GC_METHOD	The REFS_SMR_VOLUME_GC_METHOD enum specifies the garbage collection method or strategy for FSCTL_SET_REFS_SMR_VOLUME_GC_PARAMETERS.
REFS_SMR_VOLUME_GC_STATE	The REFS_SMR_VOLUME_GC_STATE enum specifies the garbage collection's current state.
RTL_MEMORY_TYPE	Defines the memory type the heap is supposed to use.
SID_NAME_USE	The SID_NAME_USE enumeration type contains values that specify the type of a security identifier (SID).
STORAGE_RESERVE_ID	Defines the storage reserve ID of a file, directory, or storage reserve area.
TOKEN_INFORMATION_CLASS	The TOKEN_INFORMATION_CLASS enumeration type contains values that specify the type of information being assigned to or retrieved from an access token.

TITLE	DESCRIPTION
TOKEN_TYPE	The TOKEN_TYPE enumeration type contains values that differentiate between a primary token and an impersonation token.
VIRTUAL_STORAGE_BEHAVIOR_CODE	Configures file system-specific behaviors used on virtual storage devices.

FILE_ACCESS_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_ACCESS_INFORMATION** structure is used to query for or set the access rights of a file.

Syntax

```
typedef struct _FILE_ACCESS_INFORMATION {
  ACCESS_MASK AccessFlags;
} FILE_ACCESS_INFORMATION, *PFILE_ACCESS_INFORMATION;
```

Members

`AccessFlags`

Flags that specify a set of access rights in the access mask of an access control entry. This member is a value of type ACCESS_MASK.

Remarks

This structure is used by the ZwQueryInformationFile routine.

The *AccessFlags* parameter is an access mask that encodes the user rights to a file object. An access mask can be used to encode the rights to an object that is assigned to a user or a user group, or to encode the requested access when a user opens an object. For more information, see ACCESS_MASK.

Requirements

Minimum supported client	Supported in Windows XP and later versions of Windows.
Header	ntifs.h (include Ntifs.h, Fltkernel.h)

See Also

ACCESS_MASK

ZwQueryInformationFile

FILE_ALL_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_ALL_INFORMATION** structure is a container for several **FILE_*XXX*_INFORMATION** structures.

Syntax

```
typedef struct _FILE_ALL_INFORMATION {
  FILE_BASIC_INFORMATION     BasicInformation;
  FILE_STANDARD_INFORMATION  StandardInformation;
  FILE_INTERNAL_INFORMATION  InternalInformation;
  FILE_EA_INFORMATION        EaInformation;
  FILE_ACCESS_INFORMATION    AccessInformation;
  FILE_POSITION_INFORMATION  PositionInformation;
  FILE_MODE_INFORMATION      ModeInformation;
  FILE_ALIGNMENT_INFORMATION AlignmentInformation;
  FILE_NAME_INFORMATION      NameInformation;
} FILE_ALL_INFORMATION, *PFILE_ALL_INFORMATION;
```

Members

`BasicInformation`

Contains basic information about the file, which includes the file attributes and the file creation time. This member is a FILE_BASIC_INFORMATION structure.

`StandardInformation`

Contains standard information about a file, which includes the file allocation size, the end-of-file offset, and whether the file is a directory. This member is a FILE_STANDARD_INFORMATION structure.

`InternalInformation`

Contains the 8-byte file reference number for the file. This member is a FILE_INTERNAL_INFORMATION structure.

`EaInformation`

Specifies the size of the extended attributes of the file. This member is a FILE_EA_INFORMATION structure.

`AccessInformation`

Specifies the client's access rights to the file. This member is a FILE_ACCESS_INFORMATION structure.

`PositionInformation`

Specifies the current file position. This member is a FILE_POSITION_INFORMATION structure.

`ModeInformation`

Specifies the access mode in which the file was created or opened. This member is a FILE_MODE_INFORMATION structure.

`AlignmentInformation`

Specifies the device's memory address alignment requirement for data transfers. This member is a

FILE_ALIGNMENT_INFORMATION structure.

`NameInformation`

Contains the file name. This member is a FILE_NAME_INFORMATION structure. This structure contains the first character in the file name string. The additional characters in the file name string immediately follow the structure. To accommodate the full file name, the buffer that is allocated to contain a **FILE_ALL_INFORMATION** structure must be large enough to contain both the structure and the part of the file name string that follows the structure.

Remarks

This structure is used by the ZwQueryInformationFile routine.

FILE_ALL_INFORMATION combines several file-information structures into a single structure to reduce the number of queries that are required to obtain information about a file.

Requirements

Minimum supported client	Supported in Windows XP and later versions of Windows.
Header	ntifs.h (include Ntifs.h, Fltkernel.h)

See Also

FILE_ACCESS_INFORMATION

FILE_ALIGNMENT_INFORMATION

FILE_BASIC_INFORMATION

FILE_EA_INFORMATION

FILE_INTERNAL_INFORMATION

FILE_MODE_INFORMATION

FILE_NAME_INFORMATION

FILE_POSITION_INFORMATION

FILE_STANDARD_INFORMATION

ZwQueryInformationFile

FILE_EA_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_EA_INFORMATION** structure is used to query for the size of the extended attributes (EA) for a file.

Syntax

```
typedef struct _FILE_EA_INFORMATION {
  ULONG EaSize;
} FILE_EA_INFORMATION, *PFILE_EA_INFORMATION;
```

Members

`EaSize`

Specifies the combined length, in bytes, of the extended attributes for the file.

Remarks

This structure is used by the ZwQueryInformationFile routine.

An extended attribute is a piece of application-specific metadata that an application can associate with a file that is not part of the file's data. In addition to the built-in attributes of a file, such as creation and modification times, applications can add non-file system attributes, such as the author's name and a description of the file content.

Requirements

Minimum supported client	Supported in Windows XP and later versions of Windows.
Header	ntifs.h (include Ntifs.h, Fltkernel.h)

See Also

ZwQueryInformationFile

FILE_MODE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_MODE_INFORMATION** structure is used to query or set the access mode of a file.

Syntax

```
typedef struct _FILE_MODE_INFORMATION {
  ULONG Mode;
} FILE_MODE_INFORMATION, *PFILE_MODE_INFORMATION;
```

Members

`Mode`

Specifies the mode in which the file will be accessed following a create-file or open-file operation. This parameter is either zero or the bitwise OR of one or more of the following file option flags:

FILE_WRITE_THROUGH

Any system services, file system drivers (FSDs), and drivers that write data to the file must actually transfer the data into the file before any requested write operation is considered complete.

FILE_SEQUENTIAL_ONLY

All accesses to the file will be sequential.

FILE_NO_INTERMEDIATE_BUFFERING

The file cannot be cached or buffered in a driver's internal buffers.

FILE_SYNCHRONOUS_IO_ALERT

All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context.

FILE_SYNCHRONOUS_IO_NONALERT

All operations on the file are performed synchronously. Wait requests in the system that must synchronize I/O queuing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context.

FILE_DELETE_ON_CLOSE

Delete the file when the last handle to the file is closed.

These flags are defined in the Wdm.h header file. For more information, see the Remarks section.

Remarks

This structure contains a set of flags that specify the mode in which the file can be accessed. These flags are a subset of the options that can be specified in the *CreateOptions* parameter of the IoCreateFile routine.

This structure is used by the ZwQueryInformationFile routine.

Requirements

Minimum supported client	Supported in Windows XP and later versions of Windows.
Header	ntifs.h (include Ntifs.h, Fltkernel.h)

See Also

IoCreateFile

ZwQueryInformationFile

MEMORY_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

Contains information about a range of pages in the virtual address space of a process. The ZwQueryVirtualMemory routine uses this structure.

Syntax

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  ULONG  AllocationProtect;
  SIZE_T RegionSize;
  ULONG  State;
  ULONG  Protect;
  ULONG  Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Members

`BaseAddress`

A pointer to the base address of the region of pages.

`AllocationBase`

A pointer to the base address of a range of allocated pages. The page pointed to by the **BaseAddress** member is contained within this allocation range.

`AllocationProtect`

The memory protection option when the region was initially allocated. This member can be one of the following constants defined in wdm.h, or 0 if the caller does not have access.

VALUE	MEANING
PAGE_NOACCESS 0x01	No access to the region of pages is allowed. An attempt to read, write, or execute within the region results in an access violation.
PAGE_EXECUTE 0x10	Execute access to the region of pages is allowed. An attempt to read or write within the region results in an access violation.

PAGE_READONLY 0x02	Read-only and execute access to the region of pages is allowed. An attempt to write within the region results in an access violation.
PAGE_READWRITE 0x04	Read, write, and execute access to the region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.
PAGE_GUARD 0x100	Read, write, and execute access to the region of pages is allowed; however, access to the region causes a "guard region entered" condition to be raised in the subject process.
PAGE_NOCACHE 0x200	Disable the placement of committed pages into the data cache.
PAGE_WRITECOMBINE 0x400	Disable the placement of committed pages into the data cache, combine the writes as well.

`RegionSize`

The size of the region in bytes beginning at the base address in which all pages have identical attributes.

`State`

The state of the pages in the region. This member can be one of the following values.

STATE	MEANING

MEM_COMMIT 0x1000	Indicates committed pages for which physical storage has been allocated, either in memory or in the paging file on disk.
MEM_FREE 0x10000	Indicates free pages not accessible to the calling process and available to be allocated.
MEM_RESERVE 0x2000	Indicates reserved pages where a range of the process's virtual address space is reserved without any physical storage being allocated.

`Protect`

The access protection of the pages in the region. This member is one of the values listed for the **AllocationProtect** member.

`Type`

The type of pages in the region. The following types are defined.

TYPE	MEANING
MEM_IMAGE 0x1000000	Indicates that the memory pages within the region are mapped into the view of an image section.
MEM_MAPPED 0x40000	Indicates that the memory pages within the region are mapped into the view of a section.

ME **M_P** **RIV** **ATE** 0x2 000 0	Indicates that the memory pages within the region are private (that is, not shared by other processes).

Requirements

Minimum supported client	Available starting with Windows 10.
Header	ntifs.h

See Also

ZwQueryVirtualMemory

MEMORY_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Defines classes of memory information that can be retrieved by using the ZwQueryVirtualMemory function.

Syntax

```
typedef enum _MEMORY_INFORMATION_CLASS {
  MemoryBasicInformation
} MEMORY_INFORMATION_CLASS;
```

Constants

MemoryBasicInformation	Memory information described in the MEMORY_BASIC_INFORMATION structure will be retrieved.

Remarks

Currently, only the **MemoryBasicInformation** value is supported for use with the ZwQueryVirtualMemory routine.

Requirements

Minimum supported client	Available starting with Windows 10.
Header	ntifs.h

See Also

ZwQueryVirtualMemory

IoGetAttachedDeviceReference function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetAttachedDeviceReference** routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.

Syntax

```
PDEVICE_OBJECT IoGetAttachedDeviceReference(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the topmost attached device object is retrieved.

Return Value

IoGetAttachedDeviceReference returns a pointer to the highest level device object in a stack of attached device objects after incrementing the reference count on the object.

Remarks

If the device object at *DeviceObject* has no device objects attached to it, *DeviceObject* and the returned pointer are equal.

Device driver writers must ensure that when they have completed all operations that required them to make this call, that they call ObDereferenceObject with the device object pointer returned by this routine. Failure to do so will prevent the system from freeing or deleting the device object because of an outstanding reference count.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	DanglingDeviceObjectReference, HwStorPortProhibitedDDIs

See Also

ObDereferenceObject

IoGetConfigurationInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetConfigurationInformation** routine returns a pointer to the I/O manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.

Syntax

```
PCONFIGURATION_INFORMATION IoGetConfigurationInformation(

);
```

Parameters

This function has no parameters.

Return Value

IoGetConfigurationInformation returns a pointer to the configuration information structure. This structure is defined as follows:

```
typedef struct _CONFIGURATION_INFORMATION {

    //
    // This field indicates the total number of disks in the system. This
    // number should be used by the driver to determine the names of new
    // disks. This field should be updated by the driver as it finds new
    // disks.
    //

    ULONG DiskCount;                // Count of hard disks thus far
    ULONG FloppyCount;              // Count of floppy disks thus far
    ULONG CdRomCount;               // Count of CD-ROM drives thus far
    ULONG TapeCount;                // Count of tape drives thus far
    ULONG ScsiPortCount;            // Count of SCSI port adapters thus far
    ULONG SerialCount;              // Count of serial devices thus far
    ULONG ParallelCount;            // Count of parallel devices thus far

    //
    // These next two fields indicate ownership of the two I/O address
    // spaces that are used by WD1003-compatible disk controllers.
    //

    BOOLEAN AtDiskPrimaryAddressClaimed;    // 0x1F0 - 0x1FF
    BOOLEAN AtDiskSecondaryAddressClaimed;  // 0x170 - 0x17F

    //
    // Indicates the structure version, as anything value beyond this will have been added.
    // Use the structure size as the version.
    //

    ULONG Version;

    //
    // Indicates the total number of medium changer devices in the system.
    // This field will be updated by the drivers as it determines that
    // new devices have been found and will be supported.
    //

    ULONG MediumChangerCount;

} CONFIGURATION_INFORMATION, *PCONFIGURATION_INFORMATION;
```

Remarks

Certain types of device drivers can use the configuration information structure's values to construct device object names with appropriate digit suffixes when each driver creates its device objects. Note that the digit suffix for device object names is a zero-based count, while the counts maintained in the configuration information structure represent the number of device objects of a particular type already created. That is, the configuration information counts are one-based.

Any driver that calls **IoGetConfigurationInformation** must increment the count for its kind of device in this structure when it creates a device object to represent a physical device.

The system-supplied SCSI port driver supplies the count of SCSI HBAs present in the computer. SCSI class drivers can read this value to determine how many HBA-specific miniport drivers might control a SCSI bus with an attached device of the class driver's type.

The configuration information structure also contains a value indicating whether an already loaded driver has claimed either of the "AT" disk I/O address ranges.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

IoAssignResources

IoQueryDeviceDescription

IoReportResourceUsage

IoSetStartIoAttributes function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetStartIoAttributes** routine sets attributes for the driver's StartIo routine.

Syntax

```
void IoSetStartIoAttributes(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        DeferredStartIo,
  BOOLEAN        NonCancelable
);
```

Parameters

`DeviceObject`

Pointer to the device object for the driver's device.

`DeferredStartIo`

If **TRUE**, the I/O manager will defer any call to the driver's StartIo routine while the driver is already inside the routine. In particular, if the *StartIo* routine calls IoStartNextPacket, the *StartIo* routine will not be called again until the current invocation completes. The default is **FALSE**.

`NonCancelable`

If **TRUE**, the IRP cannot be canceled once it has been dequeued by a call to **IoStartNextPacket**. The default is **FALSE**. Drivers that set this member to **FALSE** must synchronize their IRP handling with the cancel spin lock.

Return Value

None

Requirements

Minimum supported client	Available for Microsoft Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

DDI compliance rules	StartIoCancel, StartIoRecursion, HwStorPortProhibitedDDIs

See Also

IoStartNextPacket

StartIo

IoSizeOfIrp macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoSizeOfIrp** routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.

Syntax

```
void IoSizeOfIrp(
   StackSize
);
```

Parameters

`StackSize`

Specifies the number of stack locations for the IRP.

Return Value

IoSizeOfIrp returns the size, in bytes, of the IRP.

Remarks

The input *StackSize* value is either that of the next-lower driver's device object or one more than that value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IoAllocateIrp

IoMakeAssociatedIrp

IoStartNextPacket function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartNextPacket** routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.

Syntax

```
void IoStartNextPacket(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        Cancelable
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the IRP is to be dequeued.

`Cancelable`

Specifies whether IRPs in the device queue can be canceled.

Return Value

None

Remarks

If there are no IRPs currently in the device queue for the target *DeviceObject*, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass **TRUE** in the *Cancelable* parameter. If *Cancelable* is **TRUE**, the I/O manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a *StartIo* routine cannot call **IoStartNextPacket**.

Drivers that call **IoStartNextPacket** from their *StartIo* routine should be aware of recursion issues. If a driver can call **IoStartNextPacket** on a large number of requests in succession from its *StartIo* routine (for example, when a device error occurs and the driver is clearing out it device queue), you should set the *DeferredStartIo* attribute for the device by using IoSetStartIoAttributes routine. This attribute ensures that the next packet will not be issued until the previous *StartIo* call returns.

Callers of **IoStartNextPacket** must be running at IRQL = DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL = DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL (see Remarks section)
DDI compliance rules	IrqlDispatch, SpinLockSafe, StartIoRecursion, HwStorPortProhibitedDDIs, IrqlDispatch(storport), SpinLockSafe(storport)

See Also

DEVICE_OBJECT

IoSetStartIoAttributes

IoStartNextPacketByKey

IoStartPacket

IoStartNextPacketByKey function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartNextPacketByKey** routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.

Syntax

```
void IoStartNextPacketByKey(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        Cancelable,
  ULONG          Key
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the IRP is to be dequeued.

`Cancelable`

Specifies whether IRPs in the device queue can be canceled.

`Key`

Specifies the sort key that determines which entry to remove from the queue.

Return Value

None

Remarks

If there are no IRPs currently in the device queue for the target device object, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass **TRUE** in the *Cancelable* parameter. If *Cancelable* is **TRUE**, the I/O manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a *StartIo* routine cannot call **IoStartNextPacketByKey**.

Callers of **IoStartNextPacketByKey** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL = DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DEVICE_OBJECT

IoStartNextPacket

IoStartPacket

IoStartPacket function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartPacket** routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.

Syntax

```
void IoStartPacket(
  PDEVICE_OBJECT DeviceObject,
  PIRP           Irp,
  PULONG         Key,
  PDRIVER_CANCEL CancelFunction
);
```

Parameters

`DeviceObject`

Pointer to the target device object for the IRP.

`Irp`

Pointer to the IRP to be processed.

`Key`

Pointer to a value that determines where to insert the packet into the device queue. If this is zero, the packet is inserted at the tail of the device queue.

`CancelFunction`

Specifies the entry point for a driver-supplied Cancel routine.

Return Value

None

Remarks

If the driver is already busy processing a request for the target device object, then the packet is queued in the device queue. Otherwise, this routine calls the driver's *StartIo* routine with the specified IRP.

If a non-NULL *CancelFunction* pointer is supplied, it is set in the IRP so the driver's *Cancel* routine is called if the IRP is canceled before its completion.

Drivers that do not have a *StartIo* routine cannot call **IoStartPacket**.

Callers of **IoStartPacket** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's Dispatch routine at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

DEVICE_OBJECT

IoMarkIrpPending

IoSetCancelRoutine

IoStartNextPacket

IoStartNextPacketByKey

IoStartTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartTimer** routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.

Syntax

```
void IoStartTimer(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to a device object whose timer routine is to be called.

Return Value

None

Remarks

The driver must already have set up the IoTimer routine for the *DeviceObject* by calling **IoInitializeTimer**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Windows
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoInitializeTimer

IoStopTimer

IoTimer

KeInitializeDpc

KeInitializeTimer

KeSetTimer

IoStopTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStopTimer** routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.

Syntax

```
void IoStopTimer(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object with which the IoTimer routine is associated.

Return Value

None

Remarks

The driver-supplied IoTimer routine can be reenabled with a call to **IoStartTimer**.

Do not call **IoStopTimer** from within the *IoTimer* routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Windows
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoInitializeTimer

IoStartTimer

IoWriteErrorLogEntry function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWriteErrorLogEntry** routine queues a given error log packet to the system error logging thread.

Syntax

```
void IoWriteErrorLogEntry(
  PVOID ElEntry
);
```

Parameters

`ElEntry`

Pointer to the error log packet the driver has allocated with IoAllocateErrorLogEntry and filled in by the caller.

Return Value

None

Remarks

IoWriteErrorLogEntry frees the error log entry. Drivers must not call **IoFreeErrorLogEntry** on a log entry that they have already passed to **IoWriteErrorLogEntry**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

IO_ERROR_LOG_PACKET

IoAllocateErrorLogEntry

IoFreeErrorLogEntry

KeGetProcessorIndexFromNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetProcessorIndexFromNumber** routine converts a group number and a group-relative processor number to a systemwide processor index.

Syntax

```
ULONG KeGetProcessorIndexFromNumber(
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcNumber`

A pointer to a caller-allocated PROCESSOR_NUMBER structure that contains a group number and a group-relative processor number.

Return Value

KeGetProcessorIndexFromNumber returns a systemwide processor index if the call is successful. If *ProcNumber* points to an invalid **PROCESSOR_NUMBER** value, the routine returns INVALID_PROCESSOR_INDEX, which is defined in the Wdm.h header file.

Remarks

This routine accepts as input a **PROCESSOR_NUMBER** structure that identifies a processor by its group number and its processor number within the group. The return value is a processor index that identifies the processor across the entire multiprocessor system.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

To obtain the total number of active logical processors in the system, call the KeQueryActiveProcessorCountEx routine and set this routine's *GroupNumber* parameter to ALL_PROCESSOR_GROUPS.

The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorNumberFromIndex

KeQueryActiveProcessorCountEx

PROCESSOR_NUMBER

KeGetProcessorNumberFromIndex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetProcessorNumberFromIndex** routine converts a systemwide processor index to a group number and a group-relative processor number.

Syntax

```
NTSTATUS KeGetProcessorNumberFromIndex(
  ULONG             ProcIndex,
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcIndex`

A systemwide processor index. If a multiprocessor system contains a total of *n* logical processors, valid processor indexes range from 0 to *n*-1.

`ProcNumber`

A pointer to a caller-allocated PROCESSOR_NUMBER structure into which the routine writes the group number and group-relative processor number of the processor that is identified by *ProcIndex*.

Return Value

KeGetProcessorNumberFromIndex returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ProcIndex* parameter value is not a valid processor index.

Remarks

This routine accepts as input a processor index that identifies the processor across the entire multiprocessor system. The output value is a **PROCESSOR_NUMBER** structure that identifies a processor by its group number and its processor number within the group.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

To obtain the total number of active logical processors in the system, call the KeQueryActiveProcessorCountEx routine.

The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.

Examples

The following code example uses the **KeQueryActiveProcessorCountEx** and **KeGetProcessorNumberFromIndex** routines to enumerate all active logical processors in the system:

```
ULONG Count;
ULONG ProcIndex;
PROCESSOR_NUMBER ProcNumber;

Count = KeQueryActiveProcessorCountEx(ALL_PROCESSOR_GROUPS);
for (ProcIndex = 0; ProcIndex < Count; ProcIndex += 1)
{
    KeGetProcessorNumberFromIndex(ProcIndex, &ProcNumber);

    // Do something with the contents of ProcNumber.
    ...
}
```

The constant value ALL_PROCESSOR_GROUPS is defined in Winnt.h and Ntdef.h.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntddk.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorIndexFromNumber

KeQueryActiveProcessorCountEx

PROCESSOR_NUMBER

KeQueryPerformanceCounter function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryPerformanceCounter** routine retrieves the current value and frequency of the performance counter.

Use **KeQueryPerformanceCounter** to acquire high resolution (<1us) time stamps for time interval measurements.

Syntax

```
NTHALAPI LARGE_INTEGER KeQueryPerformanceCounter(
  PLARGE_INTEGER PerformanceFrequency
);
```

Parameters

`PerformanceFrequency`

A pointer to a variable to which **KeQueryPerformanceCounter** writes the performance counter frequency, in ticks per second. This parameter is optional and can be NULL if the caller does not need the counter frequency value.

Return Value

KeQueryPerformanceCounter returns the performance counter value in units of ticks.

Remarks

KeQueryPerformanceCounter returns a 64-bit integer that represents the current value of a high-resolution monotonically nondecreasing counter.

To obtain the frequency of the performance counter, specify a non-**NULL** pointer value for the *PerformanceFrequency* parameter. The frequency of the performance counter is fixed at system boot and is consistent across all processors. Therefore, a driver can cache the frequency of the performance counter during initialization.

For more info about this function and its usage, see Acquiring high-resolution time stamps.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
DLL	Hal.dll

IRQL	Any level

See Also

KeQueryInterruptTime

KeQuerySystemTime

KeQueryTickCount

KeQueryTimeIncrement

QueryPerformanceCounter

QueryPerformanceFrequency

KeSetKernelStackSwapEnable function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetKernelStackSwapEnable** routine enables and disables swapping of the caller's stack to disk.

Syntax

```
BOOLEAN KeSetKernelStackSwapEnable(
  BOOLEAN Enable
);
```

Parameters

`Enable`

Specifies whether to enable swapping of the stack that belongs to the calling thread. If **TRUE**, swapping is enabled and the contents of the stack can be paged in and out of memory. If **FALSE**, swapping is disabled and the stack is memory-resident.

Return Value

KeSetKernelStackSwapEnable returns a BOOLEAN value that indicates whether stack swapping was enabled at the time that the call was initiated. This value is **TRUE** if stack swapping was previously enabled and is **FALSE** if it was disabled.

Remarks

A kernel-mode driver can call this routine to control whether its stack is pageable or locked in memory.

Stack swapping can occur only if the thread is in a wait state that was caused by a request from a user-mode application. Stack swapping never occurs for wait states that are initiated by kernel-mode components, regardless of whether stack swapping is enabled.

It is not typically necessary to disable stack swapping. Do this only in rare cases. For an example that discusses alternatives to disabling stack swapping, see the following Examples section.

In a call to a kernel-mode wait routine, such as KeWaitForSingleObject, the caller specifies a *WaitMode* parameter to indicate whether the caller waits in kernel mode or user mode. If *WaitMode* = **UserMode**, and if the wait duration is sufficiently long, the memory manager might page out sections of the stack that belongs to the waiting thread. However, if the stack contains data items that must remain memory-resident for the duration of the wait, the thread can prevent the stack from being paged out by calling **KeSetKernelStackSwapEnable** and specifying *Enable* = **FALSE**.

A thread must not exit (terminate) while stack swapping is disabled or a system bug check will occur.

Examples

In the following code example, a driver thread allocates an event on its stack and calls **KeSetKernelStackSwap** to temporarily lock the stack in memory until the event is signaled. After the wait completes, the thread calls **KeSetKernelStackSwap** again, if necessary, to restore the original stack-swapping state of the thread.

```
KEVENT event;
BOOLEAN oldSwapEnable;
NTSTATUS status;

oldSwapEnable = KeSetKernelStackSwapEnable(FALSE);

KeInitializeEvent(&event, SynchronizationEvent, FALSE);

//
// TO DO: Insert code here to pass the event to another thread
// that will set the event to the signaled state.
//
...

status = KeWaitForSingleObject(&event, UserRequest, KernelMode, FALSE, NULL);

if (oldSwapEnable)
{
    KeSetKernelStackSwapEnable(TRUE);
}
```

An event object must be memory-resident while it can be set to a signaled or nonsignaled state, or while a thread waits on the event. For more information, see Defining and Using an Event Object. Frequently, the use of the **KeSetKernelStackSwap** routine is unnecessary and can be avoided by allocating only pageable data items on the stack. In the previous example, the driver thread must lock the stack because the event object is allocated on the stack. A better alternative might be to simply allocate the event from nonpaged pool.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

KeInitializeEvent

KeWaitForSingleObject

KeStallExecutionProcessor function

1/25/2019 • 2 minutes to read • Edit Online

The **KeStallExecutionProcessor** routine stalls the caller on the current processor for a specified time interval.

Syntax

```
NTHALAPI VOID KeStallExecutionProcessor(
  ULONG MicroSeconds
);
```

Parameters

`MicroSeconds`

Specifies the number of microseconds to stall.

Return Value

None

Remarks

KeStallExecutionProcessor is a processor-dependent routine that busy-waits for at least the specified number of microseconds, but not significantly longer.

This routine is for use by device drivers and other software that must wait for an interval of less than a clock tick but more than for a few instructions. If you use this routine you must minimize the stall interval, typically to less than 50 microseconds. If a driver must wait for a longer interval, you should use a different synchronization technique.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
DLL	Hal.dll
IRQL	Any level

See Also

KeDelayExecutionThread

KeWaitForMultipleObjects

KeWaitForSingleObject

NtAllocateVirtualMemory function

1/25/2019 • 6 minutes to read • Edit Online

The **NtAllocateVirtualMemory** routine reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(
  HANDLE    ProcessHandle,
  PVOID     *BaseAddress,
  ULONG_PTR ZeroBits,
  PSIZE_T   RegionSize,
  ULONG     AllocationType,
  ULONG     Protect
);
```

Parameters

`ProcessHandle`

A handle for the process for which the mapping should be done. Use the **NtCurrentProcess** macro, defined in Ntddk.h, to specify the current process.

`BaseAddress`

A pointer to a variable that will receive the base address of the allocated region of pages. If the initial value of this parameter is non-**NULL**, the region is allocated starting at the specified virtual address rounded down to the next host page size address boundary. If the initial value of this parameter is **NULL**, the operating system will determine where to allocate the region.

`ZeroBits`

The number of high-order address bits that must be zero in the base address of the section view. Used only when the operating system determines where to allocate the region, as when *BaseAddress* is **NULL**. Note that when ZeroBits is larger than 32, it becomes a bitmask.

`RegionSize`

A pointer to a variable that will receive the actual size, in bytes, of the allocated region of pages. The initial value of this parameter specifies the size, in bytes, of the region and is rounded up to the next host page size boundary. **RegionSize* cannot be zero on input.

`AllocationType`

A bitmask containing flags that specify the type of allocation to be performed. The following table describes these flags.

FLAG	MEANING
MEM_COMMIT	The specified region of pages is to be committed. One of MEM_COMMIT, MEM_RESET, or MEM_RESERVE must be set.

MEM_PHYSICAL	Allocate physical memory. This flag is solely for use with Address Windowing Extensions (AWE) memory. If MEM_PHYSICAL is set, MEM_RESERVE must also be set. No other flags may be set. If MEM_PHYSICAL is set, *Protect* must be set to PAGE_READWRITE.
MEM_RESERVE	The specified region of pages is to be reserved. One of MEM_COMMIT, MEM_RESET, or MEM_RESERVE must be set.
MEM_RESET	Reset the state of the specified region so that if the pages are in paging file, they are discarded and pages of zeros are brought in. If the pages are in memory and modified, they are marked as not modified so that they will not be written out to the paging file. The contents are not zeroed. The *Protect* parameter is not used, but it must be set to a valid value. One of MEM_COMMIT, MEM_RESET, or MEM_RESERVE must be set. If MEM_RESET is set, no other flag may be set.
MEM_TOP_DOWN	The specified region should be created at the highest virtual address possible based on *ZeroBits*.
MEM_WRITE_WATCH	The specified region should be created at the highest virtual address possible based on *ZeroBits*.

`Protect`

A bitmask containing page protection flags that specify the protection desired for the committed region of pages. The following table describes these flags.

FLAG	MEANING
PAGE_NOACCESS	No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_READONLY	Read-only and execute access to the committed region of pages is allowed. An attempt to write the committed region results in an access violation.
PAGE_READWRITE	Read, write, and execute access to the committed region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read only/copy on write.
PAGE_EXECUTE	Execute access to the committed region of pages is allowed. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Execute and read access to the committed region of pages are allowed. An attempt to write to the committed region results in an access violation.

PAGE_GUARD	Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception. Guard pages thus act as a one-shot access alarm. This flag is a page protection modifier, valid only when used with one of the page protection flags other than PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator.
PAGE_NOCACHE	The region of pages should be allocated as noncacheable. PAGE_NOCACHE is not allowed for sections.
PAGE_WRITECOMBINE	Enables write combining, that is, coalescing writes from cache to main memory, where the hardware supports it. This flag is used primarily for frame buffer memory so that writes to the same cache line are combined where possible before being written to the device. This can greatly reduce writes across the bus to (for example) video memory. If the hardware does not support write combining, the flag is ignored. This flag is a page protection modifier, valid only when used with one of the page protection flags other than PAGE_NOACCESS.

Return Value

ZwAllocateVirtualMemory returns either STATUS_SUCCESS or an error status code. Possible error status codes include the following:

STATUS_ACCESS_DENIED

STATUS_ALREADY_COMMITTED

STATUS_COMMITMENT_LIMIT

STATUS_CONFLICTING_ADDRESSES

STATUS_INSUFFICIENT_RESOURCES

STATUS_INVALID_HANDLE

STATUS_INVALID_PAGE_PROTECTION

STATUS_NO_MEMORY

STATUS_OBJECT_TYPE_MISMATCH

STATUS_PROCESS_IS_TERMINATING

Remarks

ZwAllocateVirtualMemory can perform the following operations:

- Commit a region of pages reserved by a previous call to **ZwAllocateVirtualMemory**.
- Reserve a region of free pages.
- Reserve and commit a region of free pages.

Kernel-mode drivers can use **ZwAllocateVirtualMemory** to reserve a range of application-accessible virtual addresses in the specified process and then make additional calls to **ZwAllocateVirtualMemory** to commit

individual pages from the reserved range. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

Each page in the process's virtual address space is in one of the three states described in the following table.

STATE	MEANING
FREE	The page is not committed or reserved and is not accessible to the process. **ZwAllocateVirtualMemory** can reserve, or simultaneously reserve and commit, a free page.
RESERVED	The range of addresses cannot be used by other allocation functions, but the page is not accessible to the process and has no physical storage associated with it. **ZwAllocateVirtualMemory** can commit a reserved page, but it cannot reserve it a second time. **ZwFreeVirtualMemory** can release a reserved page, making it a free page.
COMMITTED	Physical storage is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. **ZwAllocateVirtualMemory** can commit an already committed page. This means that you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. **ZwFreeVirtualMemory** can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page.

Memory allocated by calling **ZwAllocateVirtualMemory** must be freed by calling **ZwFreeVirtualMemory**.

For more information about memory management, see Memory Management for Windows Drivers.

> **Note** If the call to the **ZwAllocateVirtualMemory** function occurs in user mode, you should use the name "**NtAllocateVirtualMemory**" instead of "**ZwAllocateVirtualMemory**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwFreeVirtualMemory

NtClose function

1/25/2019 • 2 minutes to read • Edit Online

The **NtClose** routine closes an object handle.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtClose(
  HANDLE Handle
);
```

Parameters

`Handle`

Handle to an object of any type.

Return Value

NtClose returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. In particular, it returns STATUS_INVALID_HANDLE if *Handle* is not a valid handle, or STATUS_HANDLE_NOT_CLOSABLE if the calling thread does not have permission to close the handle.

Remarks

NtClose is a generic routine that operates on any type of object.

Closing an open object handle causes that handle to become invalid. The system also decrements the handle count for the object and checks whether the object can be deleted. The system does not actually delete the object until all of the object's handles are closed and no referenced pointers remain.

A driver must close every handle that it opens as soon as the handle is no longer required. Kernel handles, which are those that are opened by a system thread or by specifying the OBJ_KERNEL_HANDLE flag, can be closed only when the previous processor mode is **KernelMode**. This requirement applies both to system threads and to dispatch routines for IRPs that were issued from kernel mode. (For more information about the previous processor mode, see ExGetPreviousMode.) For example, a handle that NtCreateKey returns to a DriverEntry routine cannot subsequently be closed by the same driver's dispatch routines. A **DriverEntry** routine runs in a system process, whereas dispatch routines usually run either in the context of the thread issuing the current I/O request, or, for lower-level drivers, in an arbitrary thread context.

A nonkernel handle can be closed only if one of two conditions is met: The previous processor mode is **KernelMode**, or the calling thread has sufficient permission to close the handle. An example of the latter occurs when the calling thread is the one that created the handle.

Callers of **NtClose** should not assume that this routine automatically waits for all I/O to complete prior to returning.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtClose**" instead of "**ZwClose**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can

behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateDirectoryObject

ZwCreateFile

ZwCreateKey

ZwOpenKey

ZwOpenSection

NtCreateFile function

1/25/2019 • 16 minutes to read • Edit Online

The **NtCreateFile** routine creates a new file or opens an existing file.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateFile(
  PHANDLE            FileHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PIO_STATUS_BLOCK   IoStatusBlock,
  PLARGE_INTEGER     AllocationSize,
  ULONG              FileAttributes,
  ULONG              ShareAccess,
  ULONG              CreateDisposition,
  ULONG              CreateOptions,
  PVOID              EaBuffer,
  ULONG              EaLength
);
```

Parameters

`FileHandle`

A pointer to a HANDLE variable that receives a handle to the file.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects, the caller can specify any of the following access rights, which are specific to files.

ACCESS_MASK FLAG	ALLOWS CALLER TO DO THIS
FILE_READ_DATA	Read data from the file.
FILE_READ_ATTRIBUTES	Read the attributes of the file. (For more information, see the description of the *FileAttributes* parameter.)
FILE_READ_EA	Read the extended attributes (EAs) of the file. This flag is irrelevant for device and intermediate drivers.
FILE_WRITE_DATA	Write data to the file.
FILE_WRITE_ATTRIBUTES	Write the attributes of the file. (For more information, see the description of the *FileAttributes* parameter.)
FILE_WRITE_EA	Change the extended attributes (EAs) of the file. This flag is irrelevant for device and intermediate drivers.
FILE_APPEND_DATA	Append data to the file.

FILE_EXECUTE	Use system paging I/O to read data from the file into memory. This flag is irrelevant for device and intermediate drivers.

> **Note** Do not specify FILE_READ_DATA, FILE_WRITE_DATA, FILE_APPEND_DATA, or FILE_EXECUTE when you create or open a directory.

The caller can only specify a generic access right, GENERIC_*XXX*, for a file, not a directory. Generic access rights correspond to specific access rights as shown in the following table.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
GENERIC_READ	STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES, FILE_READ_EA, and SYNCHRONIZE.
GENERIC_WRITE	STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA, FILE_APPEND_DATA, and SYNCHRONIZE.
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, FILE_EXECUTE, FILE_READ_ATTRIBUTES, and SYNCHRONIZE. This value is irrelevant for device and intermediate drivers.
GENERIC_ALL	FILE_ALL_ACCESS.

For example, if you specify GENERIC_READ for a file object, the routine maps this value to the FILE_GENERIC_READ bitmask of specific access rights. In the preceding table, the specific access rights that are listed for GENERIC_READ correspond to the access flags that are contained in the FILE_GENERIC_READ bitmask. If the file is actually a directory, the caller can also specify the following generic access rights.

DESIREDACCESS FLAG	ALLOWS CALLER TO DO THIS
FILE_LIST_DIRECTORY	List the files in the directory.
FILE_TRAVERSE	Traverse the directory, in other words, include the directory in the path of a file.

For more information about access rights, see ACCESS_MASK. `ObjectAttributes` A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. `IoStatusBlock` A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and other information about the requested operation. In particular, the **Information** member receives one of the following values:

- FILE_CREATED
- FILE_OPENED
- FILE_OVERWRITTEN
- FILE_SUPERSEDED
- FILE_EXISTS
- FILE_DOES_NOT_EXIST

`AllocationSize` A pointer to a LARGE_INTEGER that contains the initial allocation size, in bytes, for a file that is created or overwritten. If *AllocationSize* is **NULL**, no allocation size is specified. If no file is created or overwritten, *AllocationSize* is ignored. `FileAttributes` Specifies one or more FILE_ATTRIBUTE_*XXX* flags, which represent the file attributes to set if you create or overwrite a file. The caller usually specifies FILE_ATTRIBUTE_NORMAL, which sets

the default attributes. For a list of valid FILE_ATTRIBUTE_*XXX* flags, see the CreateFile routine in the Microsoft Windows SDK documentation. If no file is created or overwritten, *FileAttributes* is ignored. `ShareAccess` Type of share access, which is specified as zero or any combination of the following flags.

***ShareAccess* flag**	**Allows other threads to do this**
FILE_SHARE_READ	Read the file
FILE_SHARE_WRITE	Write the file
FILE_SHARE_DELETE	Delete the file

Device and intermediate drivers usually set *ShareAccess* to zero, which gives the caller exclusive access to the open file. `CreateDisposition` Specifies the action to perform if the file does or does not exist. *CreateDisposition* can be one of the values in the following table.

***CreateDisposition* value**	**Action if file exists**	**Action if file does not exist**
FILE_SUPERSEDE	Replace the file.	Create the file.
FILE_CREATE	Return an error.	Create the file.
FILE_OPEN	Open the file.	Return an error.
FILE_OPEN_IF	Open the file.	Create the file.
FILE_OVERWRITE	Open the file, and overwrite it.	Return an error.
FILE_OVERWRITE_IF	Open the file, and overwrite it.	Create the file.

`CreateOptions` Specifies the options to apply when the driver creates or opens the file. Use one or more of the flags in the following table.

***CreateOptions* flag**	**Meaning**
FILE_DIRECTORY_FILE	The file is a directory. Compatible *CreateOptions* flags are FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT, and FILE_OPEN_BY_FILE_ID. The *CreateDisposition* parameter must be set to FILE_CREATE, FILE_OPEN, or FILE_OPEN_IF.
FILE_NON_DIRECTORY_FILE	The file is not a directory. The file object to open can represent a data file; a logical, virtual, or physical device; or a volume.
FILE_WRITE_THROUGH	System services, file-system drivers, and drivers that write data to the file must actually transfer the data to the file before any requested write operation is considered complete.
FILE_SEQUENTIAL_ONLY	All access to the file will be sequential.
FILE_RANDOM_ACCESS	Access to the file can be random, so no sequential read-ahead operations should be performed by file-system drivers or by the system.

FILE_NO_INTERMEDIATE_BUFFERING	The file cannot be cached or buffered in a driver's internal buffers. This flag is incompatible with the *DesiredAccess* parameter's FILE_APPEND_DATA flag.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file-position pointer. If this flag is set, the SYNCHRONIZE flag must be set in the *DesiredAccess* parameter.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the file are performed synchronously. Waits in the system that synchronize I/O queuing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file-position context. If this flag is set, the SYNCHRONIZE flag must be set in the *DesiredAccess* parameter.
FILE_CREATE_TREE_CONNECTION	Create a tree connection for this file in order to open it over the network. This flag is not used by device and intermediate drivers.
FILE_COMPLETE_IF_OPLOCKED	Complete this operation immediately with an alternate success code of STATUS_OPLOCK_BREAK_IN_PROGRESS if the target file is oplocked, rather than blocking the caller's thread. If the file is oplocked, another caller already has access to the file. This flag is not used by device and intermediate drivers.
FILE_NO_EA_KNOWLEDGE	If the extended attributes (EAs) for an existing file being opened indicate that the caller must understand EAs to properly interpret the file, **NtCreateFile** should return an error. This flag is irrelevant for device and intermediate drivers.
FILE_OPEN_REPARSE_POINT	Open a file with a reparse point and bypass normal reparse point processing for the file. For more information, see the following Remarks section.
FILE_DELETE_ON_CLOSE	The system deletes the file when the last handle to the file is passed to NtClose. If this flag is set, the DELETE flag must be set in the *DesiredAccess* parameter.

FILE_OPEN_BY_FILE_ID	The file name that is specified by the *ObjectAttributes* parameter includes a binary 8-byte or 16-byte file reference number or object ID for the file, depending on the file system as shown below. Optionally, a device name followed by a backslash character may proceed these binary values. For example, a device name will have the following format.
	
	`??\C:<8 bytes of binary FileID>`
	`\device\HardDiskVolume1<16 bytes of binary ObjectID>`
	
	• On NTFS, this can be a 8-byte or 16-byte reference number or object ID. A 16-byte reference number is the same as an 8-byte number padded with zeros.
	• On ReFS, this can be an 8-byte or 16-byte reference number. A 16-byte number is not related to an 8-byte number. Object IDs are not supported.
	• The FAT, ExFAT, UDFS, and CDFS file systems do not support this flag.
	
	This number is assigned by and specific to the particular file system.
	
	Note Because the filename field will partly contain a binary blob, it is incorrect to assume that this is a valid Unicode string, and more importantly may not be a null terminated string.

FILE_OPEN_FOR_BACKUP_INTENT	The file is being opened for backup intent. Therefore, the system should check for certain access rights and grant the caller the appropriate access to the file—before checking the *DesiredAccess* parameter against the file's security descriptor. This flag not used by device and intermediate drivers.
FILE_RESERVE_OPFILTER	This flag allows an application to request a Filter opportunistic lock (oplock) to prevent other applications from getting share violations. If there are already open handles, the create request will fail with STATUS_OPLOCK_NOT_GRANTED. For more information, see the following Remarks section.
FILE_OPEN_REQUIRING_OPLOCK	The file is being opened and an opportunistic lock (oplock) on the file is being requested as a single atomic operation. The file system checks for oplocks before it performs the create operation, and will fail the create with a return code of STATUS_CANNOT_BREAK_OPLOCK if the result would be to break an existing oplock. **Note** The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2 and later Windows operating systems.
FILE_SESSION_AWARE	The client opening the file or device is session aware and per session access is validated if necessary. **Note** The FILE_SESSION_AWARE flag is available starting withWindows 8.

`EaBuffer` For device and intermediate drivers, this parameter must be a **NULL** pointer. `EaLength` For device and intermediate drivers, this parameter must be zero. ## Return Value **NtCreateFile** returns STATUS_SUCCESS on success or an appropriate NTSTATUS error code on failure. In the latter case, the caller can determine the cause of the failure by checking the *IoStatusBlock* parameter.

> **Note NtCreateFile** might return STATUS_FILE_LOCK_CONFLICT as the return value or in the **Status** member of the **IO_STATUS_BLOCK** structure that is pointed to by the *IoStatusBlock* parameter. This would occur only if the NTFS log file is full, and an error occurs while **NtCreateFile** tries to handle this situation.

Remarks

NtCreateFile supplies a handle that the caller can use to manipulate a file's data, or the file object's state and attributes. For more information, see Using Files in a Driver.

Once the handle pointed to by *FileHandle* is no longer in use, the driver must call NtClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

There are two alternate ways to specify the name of the file to be created or opened with **NtCreateFile**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *ObjectAttributes*.
2. As pathname relative to the directory file represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*.

Setting certain flags in the *DesiredAccess* parameter results in the following effects:

- For a caller to synchronize an I/O completion by waiting for the returned *FileHandle*, the SYNCHRONIZE flag must be set. Otherwise, a caller that is a device or intermediate driver must synchronize an I/O completion by using an event object.
- If the caller sets only the FILE_APPEND_DATA and SYNCHRONIZE flags, it can write only to the end of the file, and any offset information about write operations to the file is ignored. The file will automatically be extended as necessary for this type of operation.
- Setting the FILE_WRITE_DATA flag for a file also allows the caller to write beyond the end of the file. Again, the file is automatically extended as necessary.
- If the caller sets only the FILE_EXECUTE and SYNCHRONIZE flags, it cannot directly read or write any data to the file using the returned *FileHandle*. That is, all operations on the file occur through the system pager in response to instruction and data-access operations. Device and intermediate drivers should not set the FILE_EXECUTE flag.

The *ShareAccess* parameter determines whether separate threads can access the same file, possibly simultaneously. Provided that both callers have the appropriate access privileges, the file can be successfully opened and shared. If the original caller of **NtCreateFile** does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, no other caller can open the file—that is, the original caller is granted exclusive access.

To successfully open a shared file, the *DesiredAccess* flags must be compatible with the *DesiredAccess* and *ShareAccess* flags of all the previous open operations that have not yet been released through . That is, the *DesiredAccess* specified to **NtCreateFile** for a given file must not conflict with the accesses that other openers of the file have disallowed.

The *CreateDisposition* value FILE_SUPERSEDE requires that the caller have DELETE access to a existing file object. If so, a successful call to **NtCreateFile** with FILE_SUPERSEDE on an existing file effectively deletes that file, and

then recreates it. This implies that, if the file has already been opened by another thread, it opened the file by specifying a *ShareAccess* parameter with the FILE_SHARE_DELETE flag set. Note that this type of disposition is consistent with the POSIX style of overwriting files.

The *CreateDisposition* values FILE_OVERWRITE_IF and FILE_SUPERSEDE are similar. If **NtCreateFile** is called with a existing file and either of these *CreateDisposition* values, the file will be replaced.

Overwriting a file is semantically equivalent to a supersede operation, except for the following:

- The caller must have write access to the file, rather than delete access. This implies that, if the file has already been opened by another thread, it opened the file with the FILE_SHARE_WRITE flag set in the input *ShareAccess*.
- The specified file attributes are logically ORed with those already on the file. This implies that, if the file has already been opened by another thread, a subsequent caller of **NtCreateFile** cannot disable existing *FileAttributes* flags but can enable additional flags for the same file. Note that this style of overwriting files is consistent with MS-DOS, Microsoft Windows 3.1, and OS/2.

The FILE_DIRECTORY_FILE *CreateOptions* value specifies that the file to be created or opened is a directory. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the given file to be opened is not a directory file, or if the caller specified an inconsistent *CreateOptions* or *CreateDisposition* value, the call to **NtCreateFile** will fail.

The FILE_NO_INTERMEDIATE_BUFFERING *CreateOptions* flag prevents the file system from performing any intermediate buffering on behalf of the caller. Specifying this flag places the following restrictions on the caller's parameters to other **Zw*Xxx*File** routines.

- Any optional *ByteOffset* passed to NtReadFile or NtWriteFile must be a multiple of the sector size.
- The *Length* passed to **NtReadFile** or **NtWriteFile** must be an integral of the sector size. Note that specifying a read operation to a buffer whose length is exactly the sector size might result in a lesser number of significant bytes being transferred to that buffer if the end of the file was reached during the transfer.
- Buffers must be aligned in accordance with the alignment requirement of the underlying device. To obtain this information, call **NtCreateFile** to get a handle for the file object that represents the physical device, and pass that handle to NtQueryInformationFile. For a list of the system's FILE_*XXX*_ALIGNMENT values, see DEVICE_OBJECT.
- Calls to NtSetInformationFile with the *FileInformationClass* parameter set to **FilePositionInformation** must specify an offset that is a multiple of the sector size.

The FILE_SYNCHRONOUS_IO_ALERT and FILE_SYNCHRONOUS_IO_NONALERT *CreateOptions* flags, which are mutually exclusive as their names suggest, specify that all I/O operations on the file will be synchronous—as long as they occur through the file object referred to by the returned *FileHandle*. All I/O on such a file is serialized across all threads using the returned handle. If either of these *CreateOptions* flags is set, the SYNCHRONIZE *DesiredAccess* flag must also be set—to compel the I/O manager to use the file object as a synchronization object. In these cases, the I/O manager keeps track of the current file-position offset, which you can pass to **NtReadFile** and **NtWriteFile**. Call **NtQueryInformationFile** or **NtSetInformationFile** to get or set this position.

If the *CreateOptions* FILE_OPEN_REPARSE_POINT flag is not specified and **NtCreateFile** attempts to open a file with a reparse point, normal reparse point processing occurs for the file. If, on the other hand, the FILE_OPEN_REPARSE_POINT flag is specified, normal reparse processing does not occur and **NtCreateFile** attempts to directly open the reparse point file. In either case, if the open operation was successful, **NtCreateFile** returns STATUS_SUCCESS; otherwise, the routine returns an NTSTATUS error code. **NtCreateFile** never returns STATUS_REPARSE.

The *CreateOptions* FILE_OPEN_REQUIRING_OPLOCK flag eliminates the time between when you open the file and request an oplock that could potentially allow a third party to open the file and get a sharing violation. An application can use the FILE_OPEN_REQUIRING_OPLOCK flag on **NtCreateFile** and then request any oplock. This ensures that an oplock owner will be notified of any subsequent open request that causes a sharing violation.

In Windows 7, if other handles exist on the file when an application uses the FILE_OPEN_REQUIRING_OPLOCK flag, the create operation will fail with STATUS_OPLOCK_NOT_GRANTED. This restriction no longer exists starting with Windows 8.

If this create operation would break an oplock that already exists on the file, then setting the FILE_OPEN_REQUIRING_OPLOCK flag will cause the create operation to fail with STATUS_CANNOT_BREAK_OPLOCK. The existing oplock will not be broken by this create operation.

An application that uses the FILE_OPEN_REQUIRING_OPLOCK flag must request an oplock after this call succeeds, or all subsequent attempts to open the file will be blocked without the benefit of normal oplock processing. Similarly, if this call succeeds but the subsequent oplock request fails, an application that uses this flag must close its handle after it detects that the oplock request has failed.

> **Note** The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2 and later Windows operating systems. The Microsoft file systems that implement this flag in Windows 7 are NTFS, FAT, and exFAT.

The *CreateOptions* flag FILE_RESERVE_OPFILTER allows an application to request a Level 1, Batch, or Filter oplock to prevent other applications from getting share violations. However, FILE_RESERVE_OPFILTER is only practically useful for Filter oplocks. To use it, you must complete the following steps:

1. Issue a create request with *CreateOptions* of FILE_RESERVE_OPFILTER, *DesiredAccess* of exactly FILE_READ_ATTRIBUTES, and *ShareAccess* of exactly FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE.
    - If there are already open handles, the create request fails with STATUS_OPLOCK_NOT_GRANTED, and the next requested oplock also fails.
    - If you open with more access or less sharing will also cause a failure of STATUS_OPLOCK_NOT_GRANTED.
2. If the create request succeeds, request an oplock.
3. Open another handle to the file to do I/O.

Step three makes this practical only for Filter oplocks. The handle opened in step 3 can have a *DesiredAccess* that contains a maximum of FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | FILE_READ_DATA | FILE_READ_EA | FILE_EXECUTE | SYNCHRONIZE | READ_CONTROL and still not break a Filter oplock. However, any *DesiredAccess* greater than FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | SYNCHRONIZE will break a Level 1 or Batch oplock and make the FILE_RESERVE_OPFILTER flag useless for those oplock types.

NTFS is the only Microsoft file system that implements FILE_RESERVE_OPFILTER.

Callers of **NtCreateFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateFile**" instead of "**NtCreateFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

DEVICE_OBJECT

IO_STATUS_BLOCK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

NtClose

NtOpenFile

NtQueryInformationFile

NtReadFile

NtSetInformationFile

NtWriteFile

NtCreateSection function

1/25/2019 • 3 minutes to read • Edit Online

The **NtCreateSection** routine creates a section object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateSection(
  PHANDLE            SectionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PLARGE_INTEGER     MaximumSize,
  ULONG              SectionPageProtection,
  ULONG              AllocationAttributes,
  HANDLE             FileHandle
);
```

Parameters

`SectionHandle`

Pointer to a HANDLE variable that receives a handle to the section object.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects (see ACCESS_MASK), the caller can specify any of the following access rights, which are specific to section objects:

***DESIREDACCESS* FLAG**	**ALLOWS CALLER TO DO THIS**
SECTION_EXTEND_SIZE	Dynamically extend the size of the section.
SECTION_MAP_EXECUTE	Execute views of the section.
SECTION_MAP_READ	Read views of the section.
SECTION_MAP_WRITE	Write views of the section.
SECTION_QUERY	Query the section object for information about the section. Drivers should set this flag.
SECTION_ALL_ACCESS	All of the previous flags combined with STANDARD_RIGHTS_REQUIRED.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`MaximumSize`

Specifies the maximum size, in bytes, of the section. **NtCreateSection** rounds this value up to the nearest multiple of PAGE_SIZE. If the section is backed by the paging file, *MaximumSize* specifies the actual size of the section. If the section is backed by an ordinary file, *MaximumSize* specifies the maximum size that the file can be extended or mapped to.

`SectionPageProtection`

Specifies the protection to place on each page in the section. Use one of the following four values: PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, or PAGE_WRITECOPY. For a description of these values, see CreateFileMapping.

`AllocationAttributes`

Specifies a bitmask of SEC_*XXX* flags that determines the allocation attributes of the section. For a description of these flags, see CreateFileMapping.

`FileHandle`

Optionally specifies a handle for an open file object. If the value of *FileHandle* is **NULL**, the section is backed by the paging file. Otherwise, the section is backed by the specified file.

Return Value

NtCreateSection returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_FILE_LOCKED	The file specified by the *FileHandle* parameter is locked.
STATUS_INVALID_FILE_FOR_SECTION	The file specified by *FileHandle* does not support sections.

STA TUS _IN VAL ID_ PA GE_ PR OTE CTI ON	The value specified for the *SectionPageProtection* parameter is invalid.
STA TUS _M APP ED_ FILE _SIZ E_Z ERO	The size of the file specified by *FileHandle* is zero, and *MaximumSize* is zero.
STA TUS _SE CTI ON_ TO O_B IG	The value of *MaximumSize* is too big. This occurs when either *MaximumSize* is greater than the system-defined maximum for sections, or if *MaximumSize* is greater than the specified file and the section is not writable.

Remarks

Once the handle pointed to by *SectionHandle* is no longer in use, the driver must call NtClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about setting up mapped sections and views of memory, see Sections and Views.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateSection**" instead of "**ZwCreateSection**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ACCESS_MASK

CreateFileMapping

InitializeObjectAttributes

ZwClose

ZwMapViewOfSection

ZwOpenSection

ZwUnmapViewOfSection

NtDeviceIoControlFile function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwDeviceIoControlFile** routine sends a control code directly to a specified device driver, causing the corresponding driver to perform the specified operation.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtDeviceIoControlFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  ULONG            IoControlCode,
  PVOID            InputBuffer,
  ULONG            InputBufferLength,
  PVOID            OutputBuffer,
  ULONG            OutputBufferLength
);
```

Parameters

`FileHandle`

Handle returned by ZwCreateFile or ZwOpenFile for the file object representing the device to which the control information should be given or from which information should be returned. The file object must have been opened for asynchronous I/O if the caller specifies an *Event*, *ApcRoutine*, and an APC context (in *ApcContext*), or a completion context (in *ApcContext*). For I/O to an underlying mass-storage device, the file object must have been opened for Direct Access to Storage Device (DASD) access.

`Event`

Handle for a caller-created event. If this parameter is supplied, the caller will be put into a wait state until the requested operation is completed and the given event is set to the Signaled state. This parameter is optional and can be **NULL**. It must be **NULL** if the caller will wait for the *FileHandle* to be set to the Signaled state.

`ApcRoutine`

Address of an optional, caller-supplied APC routine to be called when the requested operation completes. This parameter can be **NULL**. It must be **NULL** if there is an I/O completion object associated with the file object.

`ApcContext`

Pointer to a caller-determined context area. This parameter value is used as the APC context if the caller supplies an APC, or is used as the completion context if an I/O completion object has been associated with the file object. When the operation completes, either the APC context is passed to the APC, if one was specified, or the completion context is included as part of the completion message that the I/O Manager posts to the associated I/O completion object.

This parameter is optional and can be **NULL**. It must be **NULL** if *ApcRoutine* is **NULL** and there is no I/O completion object associated with the file object.

`IoStatusBlock`

Pointer to a variable that receives the final completion status and information about the operation. For successful calls that return data, the number of bytes written to the *OutputBuffer* is returned in the **Information** member.

`IoControlCode`

IOCTL_*XXX* code that indicates which device I/O control operation is to be carried out on, usually by the underlying device driver. The value of this parameter determines the format and required length of the *InputBuffer* and *OutputBuffer*, as well as which of the following parameter pairs are required. For detailed information about the system-defined, device-type-specific IOCTL_*XXX* codes, see the device technology-specific section of the Microsoft Windows Driver Kit (WDK) documentation and Device Input and Output Control Codes in the Microsoft Windows SDK documentation.

`InputBuffer`

Pointer to a caller-allocated input buffer that contains device-specific information to be given to the target device. If *IoControlCode* specifies an operation that does not require input data, this pointer can be **NULL**.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. If *InputBuffer* is **NULL**, set *InputBufferLength* to zero.

`OutputBuffer`

Pointer to a caller-allocated output buffer in which information is returned from the target device. If *IoControlCode* specifies an operation that does not produce output data, this pointer can be **NULL**.

`OutputBufferLength`

Size, in bytes, of the buffer at *OutputBuffer*. If *OutputBuffer* is **NULL**, set *OutputBufferLength* to zero.

Return Value

ZwDeviceIoControlFile returns STATUS_SUCCESS if the underlying driver(s) successfully carried out the requested operation. Otherwise, the return value can be an error status code propagated from an underlying driver. Possible error status codes include the following:

Remarks

ZwDeviceIoControlFile provides a consistent view of the input and output data to the system and to kernel-mode drivers, while providing applications and underlying drivers with a device-dependent method of specifying a communications interface.

For more information about system-defined IOCTL_*XXX* codes, and about defining driver-specific IOCTL_*XXX* or FSCTL_*XXX* values, see Using I/O Control Codes in the *Kernel Mode Architecture Guide* and Device Input and Output Control Codes in the Microsoft Windows SDK documentation.

If the caller opened the file for asynchronous I/O (with neither FILE_SYNCHRONOUS_*XXX* create/open option set), the specified event, if any, will be set to the signaled state when the device control operation completes. Otherwise, the file object specified by *FileHandle* will be set to the signaled state. If an *ApcRoutine* was specified, it is called with the *ApcContext* and *IoStatusBlock* pointers.

Minifilters should use FltDeviceIoControlFile instead of **ZwDeviceIoControlFile**.

Callers of **ZwDeviceIoControlFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **ZwDeviceIoControlFile** function occurs in user mode, you should use the name "**NtDeviceIoControlFile**" instead of "**ZwDeviceIoControlFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltDeviceIoControlFile

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoCallDriver

Using I/O Control Codes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateFile

ZwOpenFile

NtDuplicateToken function

1/25/2019 • 6 minutes to read • Edit Online

The **NtDuplicateToken** function creates a handle to a new access token that duplicates an existing token. This function can create either a primary token or an impersonation token.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtDuplicateToken(
  HANDLE             ExistingTokenHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  BOOLEAN            EffectiveOnly,
  TOKEN_TYPE         TokenType,
  PHANDLE            NewTokenHandle
);
```

Parameters

`ExistingTokenHandle`

A handle to an existing access token that was opened with the TOKEN_DUPLICATE access right. This parameter is required and cannot be **NULL**.

`DesiredAccess`

Bitmask that specifies the requested access rights for the new token. **NtDuplicateToken** compares the requested access rights with the existing token's discretionary access control list (DACL) to determine which rights are granted or denied to the new token. To request the same access rights as the existing token, specify zero. To request all access rights that are valid for the caller, specify MAXIMUM_ALLOWED. This parameter is optional and can either be zero, MAXIMUM_ALLOWED, or a bitwise OR combination of one or more of the following values:

VALUE	MEANING
DELETE	Required to delete the object.
READ_CONTROL	Required to read the DACL and ownership information for the object. For access to the system access control list (SACL), see ACCESS_SYSTEM_SECURITY later in this table.
WRITE_DAC	Required to change the DACL information for the object.
WRITE_OWNER	Required to change the ownership information in the object's security descriptor (SECURITY_DESCRIPTOR).
ACCESS_SYSTEM_SECURITY	Required to get or set the SACL in an object's ACL. The operating system grants this right to the new token only if the SE_SECURITY_NAME privilege is enabled in the access token of the calling thread.
STANDARD_RIGHTS_READ	Currently defined to equal READ_CONTROL.

STANDARD_RIGHTS_WRITE	Currently defined to equal READ_CONTROL.
STANDARD_RIGHTS_EXECUTE	Currently defined to equal READ_CONTROL.
STANDARD_RIGHTS_REQUIRED	Combines DELETE, READ_CONTROL, WRITE_DAC, and WRITE_OWNER access.
STANDARD_RIGHTS_ALL	Combines DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER, and SYNCHRONIZE access. However, the SYNCHRONIZE value is not applicable to token objects. Thus, STANDARD_RIGHTS_ALL has a functionally equivalent to STANDARD_RIGHTS_REQUIRED.
TOKEN_ADJUST_DEFAULT	Required to change the default owner, primary group, or DACL of an access token.
TOKEN_ADJUST_GROUPS	Required to adjust the attributes of the groups in an access token.
TOKEN_ADJUST_PRIVILEGES	Required to enable or disable the privileges in an access token.
TOKEN_ADJUST_SESSIONID	Required to adjust the session ID (SID) of an access token. The operating system grants this right to the new token only if the SE_TCB_NAME privilege is enabled in the access token of the calling thread.
TOKEN_ASSIGN_PRIMARY	Required to attach a primary token to a process. The operating system grants this right to the new token only if the SE_ASSIGNPRIMARYTOKEN_NAME privilege is enabled in the access token of the calling thread.
TOKEN_DUPLICATE	Required to duplicate an access token. Note that the given *ExistingTokenHandle* token must contain this right in order to successfully use this routine.
TOKEN_EXECUTE	Combines STANDARD_RIGHTS_EXECUTE and TOKEN_IMPERSONATE.
TOKEN_IMPERSONATE	Required to attach an impersonation access token to a process.
TOKEN_QUERY	Required to query an access token.
TOKEN_QUERY_SOURCE	Required to query the source of an access token.
TOKEN_READ	Combines STANDARD_RIGHTS_READ and TOKEN_QUERY.
TOKEN_WRITE	Combines STANDARD_RIGHTS_WRITE, TOKEN_ADJUST_PRIVILEGES, TOKEN_ADJUST_GROUPS, and TOKEN_ADJUST_DEFAULT.
TOKEN_ALL_ACCESS	Combines all possible token access permissions for a token.

For additional information, see Access Rights for Access-Token Objects in the Microsoft Windows SDK. Note that access tokens do not support the SYNCHRONIZE right.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that describes the requested properties for the new token. The *ObjectAttributes* parameter is optional and can be **NULL**. If the *ObjectAttributes* parameter is **NULL** or if the **SecurityDescriptor** member of the structure pointed to by the *ObjectAttributes* parameter is **NULL**, the new token receives a default security descriptor and the new token handle cannot be inherited. In that case, this default security descriptor is created from the user group, primary group, and DACL information that is stored in the caller's token.

When the *TokenType* parameter is set to **TokenImpersonation**:

- The *ObjectAttributes*parameter may be used to specify the impersonation level of the new token. This can be accomplished by setting *ObjectAttributes*->SecurityQualityOfService.ImpersonationLevel to an appropriate SECURITY_IMPERSONATION_LEVEL enumeration value. For more information, see SECURITY_QUALITY_OF_SERVICE in the Microsoft Windows SDK documentation.
- If the existing token is an impersonation token and the *ObjectAttributes* parameter provides no impersonation information, the new token's impersonation level is set to the existing token's impersonation level.
- If the existing token is a primary token and no impersonation level information is provided, the new impersonation token will have a SECURITY_IMPERSONATION_LEVEL impersonation level.

`EffectiveOnly`

A Boolean value that indicates whether the entire existing token should be duplicated into the new token or just the effective (currently enabled) part of the token. If set to **TRUE**, only the currently enabled parts of the source token will be duplicated. If set to **FALSE**, the entire existing token will be duplicated. This provides a means for a caller of a protected subsystem to limit which optional groups and privileges are made available to the protected subsystem. For example, if *EffectiveOnly* is **TRUE**, the caller could duplicate a token but remove the Administrators group and the SeTcbPrivilege right. The resulting token could then be passed to a child process (CreateProcessAsUser), which would restrict what the child process can do. This parameter is required.

`TokenType`

Specifies one of the following values from the TOKEN_TYPE enumeration.

VALUE	MEANING
TokenPrimary	The new token is a primary token. If the existing token is an impersonation token, the existing impersonation token must have an impersonation level (as provided by the *ObjectAttributes* parameter) of **SecurityImpersonation** or **SecurityDelegation**. Otherwise, **NtDuplicateToken** returns STATUS_BAD_IMPERSONATION_LEVEL is returned.
TokenImpersonation	The new token is an impersonation token. If the existing token is an impersonation token, the requested impersonation level (as provided by the *ObjectAttributes* parameter) of the new token must not be greater than the impersonation level of the existing token. Otherwise, **NtDuplicateToken** returns STATUS_BAD_IMPERSONATION_LEVEL.

The *TokenType* parameter is required and cannot be **NULL**.

`NewTokenHandle`

A pointer to a caller-allocated variable, of type HANDLE, that receives a handle to the new token. This parameter is required and cannot be **NULL**.

Return Value

NtDuplicateToken returns STATUS_SUCCESS if the call is successfull. Possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_ACCESS_VIOLATION	A memory access violation occurred. For example, if the previous mode was user-mode and invalid user-mode memory was provided, **NtDuplicateToken** returns STATUS_ACCESS_VIOLATION.
STATUS_INSUFFICIENT_RESOURCES	Sufficient memory could not be allocated to duplicate the new token.
STATUS_INVALID_PARAMETER	An invalid parameter was detected.
STATUS_BAD_IMPERSONATION_LEVEL	The requested impersonation level for the new token is greater than the impersonation level of the existing token.

STATUS_ACCESS_DENIED	**NtDuplicateToken** returns STATUS_ACCESS_DENIED if it couldn't access *ExistingTokenHandle*. This would occur if the existing token does not have the TOKEN_DUPLICATE access right.
STATUS_INVALID_HANDLE	**NtDuplicateToken** returns STATUS_INVALID_HANDLE if *ExistingTokenHandle* refers to an invalid handle.

Remarks

If no impersonation level information was provided by the *ObjectAttributes* parameter, the existing token's impersonation level will be used for the new token.

With regard to the structure pointed to by the optional *ObjectAttributes* parameter, the **SecurityQualityOfService** member of OBJECT_ATTRIBUTES points to a structure of type **SECURITY_QUALITY_OF_SERVICE**. See SECURITY_QUALITY_OF_SERVICE in the Microsoft Windows SDK documentation for information on the members of this structure.

> **Note** The **SecurityQualityOfService** member must be set after calling the InitializeObjectAttributes macro because **InitializeObjectAttributes** currently sets **SecurityQualityOfService** to **NULL**.

For information on the user-mode analog of **NtDuplicateToken**, see DuplicateTokenEx in the Windows SDK documentation.

When you have finished using the new token, call the NtClose function to close the token handle.

> **Note** If the call to the **NtDuplicateToken** function occurs in user mode, you should use the name "**NtDuplicateToken**" instead of "**NtDuplicateToken**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

SECURITY_IMPERSONATION_LEVEL

NtFlushBuffersFileEx function

1/25/2019 • 2 minutes to read • Edit Online

The **NtFlushBuffersFileEx** routine is called by a file system filter driver to send a flush request for a given file to the file system. An optional flush operation flag can be set to control how file data is written to storage.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtFlushBuffersFileEx(
  HANDLE           FileHandle,
  ULONG            Flags,
  PVOID            Parameters,
  ULONG            ParametersSize,
  PIO_STATUS_BLOCK IoStatusBlock
);
```

Parameters

`FileHandle`

Handle returned by NtCreateFile or NtOpenFile for the file whose buffers will be flushed. This parameter is required and cannot be **NULL**.

`Flags`

Flush operation flags. *Flags* can be 0 or one of the following values.

VALUE	MEANING
FLUSH_FLAGS_FILE_DATA_ONLY	If the file is on an NTFS file system, file data in the file cache will be written. No metadata is written and the underlying storage is not synchronized to flush its cache. This flag is not valid with volume handles.
FLUSH_FLAGS_NO_SYNC	If the file is on an NTFS file system, file data and metadata in the file cache will be written. The underlying storage is not synchronized to flush its cache. This flag is not valid with volume handles.

`Parameters`

Address of the caller's I/O status block. This parameter is required and cannot be **NULL**.

`ParametersSize`

The size, in bytes, of the parameters block.

`IoStatusBlock`

Address of the caller's I/O status block. This parameter is required and cannot be **NULL**.

Return Value

NtFlushBuffersFileEx returns **STATUS_SUCCESS** or an appropriate **NTSTATUS** value, such as one of the following:

RETURN CODE	DESCRIPTION
STATUS_MEDIA_WRITE_PROTECTED	The file resides on a write-protected volume; this is an error code.
STATUS_VOLUME_DISMOUNTED	The file resides on a volume that is not currently mounted; this is an error code.
STATUS_ACCESS_DENIED	The file does has neither write or append access.

Remarks

A file system filter driver can call **NtFlushBuffersFileEx** to issue an IRP_MJ_FLUSH_BUFFERS request to the file system for a given file. The flush operation is synchronous.

Minifilter drivers should call FltFlushBuffers instead of calling **NtFlushBuffersFileEx**.

Callers of **NtFlushBuffersFileEx** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **NtFlushBuffersFileEx** function occurs in user mode, you should use the name "**NtFlushBuffersFileEx**" instead of "**NtFlushBuffersFileEx**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (See Remarks section.)

See Also

FltFlushBuffers

IRP_MJ_FLUSH_BUFFERS

Using Nt and Zw Versions of the Native System Services Routines

NtCreateFile

NtOpenFile

NtFreeVirtualMemory function

1/25/2019 • 5 minutes to read • Edit Online

The **NtFreeVirtualMemory** routine releases, decommits, or both, a region of pages within the virtual address space of a specified process.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtFreeVirtualMemory(
  HANDLE  ProcessHandle,
  PVOID   *BaseAddress,
  PSIZE_T RegionSize,
  ULONG   FreeType
);
```

Parameters

`ProcessHandle`

A handle for the process in whose context the pages to be freed reside. Use the **NtCurrentProcess** macro, defined in Ntddk.h, to specify the current process.

`BaseAddress`

A pointer to a variable that will receive the virtual address of the freed region of pages.

If the MEM_RELEASE flag is set in the *FreeType* parameter, *BaseAddress* must be the base address returned by NtAllocateVirtualMemory when the region was reserved.

`RegionSize`

A pointer to a variable that will receive the actual size, in bytes, of the freed region of pages. The routine rounds the initial value of this variable up to the next host page size boundary and writes the rounded value back to this variable.

If the MEM_RELEASE flag is set in the *FreeType* parameter, the variable pointed to by *RegionSize* must be zero. **NtFreeVirtualMemory** frees the entire region that was reserved in the initial allocation call to **NtAllocateVirtualMemory**.

If the MEM_DECOMMIT flag is set in the *FreeType* parameter, **NtFreeVirtualMemory** decommits all memory pages that contain one or more bytes in the range from the *BaseAddress* parameter to (*BaseAddress* + **RegionSize*). This means, for example, that if a two-byte region of memory straddles a page boundary, both pages are decommitted.

NtFreeVirtualMemory decommits the entire region that was reserved by **NtAllocateVirtualMemory**. If the following three conditions are met, the entire region enters the reserved state:

- The MEM_DECOMMIT flag is set.
- *BaseAddress* is the base address returned by **NtAllocateVirtualMemory** when the region was reserved.
- **RegionSize* is zero.

`FreeType`

A bitmask that contains flags that describe the type of free operation that **NtFreeVirtualMemory** will perform for the specified region of pages. The possible values are listed in the following table.

***FREETYPE* FLAGS**	**DESCRIPTION**
MEM_DECOMMIT	**NtFreeVirtualMemory** will decommit the specified region of pages. The pages enter the reserved state. **NtFreeVirtualMemory** does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining their current commitment state.
MEM_RELEASE	**NtFreeVirtualMemory** will release the specified region of pages. The pages enter the free state. If you specify this flag, * *RegionSize* must be zero, and *BaseAddress*must point to the base address returned by NtAllocateVirtualMemory when the region was reserved. **NtFreeVirtualMemory** fails if either of these conditions is not met. If any pages in the region are currently committed, **NtFreeVirtualMemory** first decommits and then releases them. **NtFreeVirtualMemory** does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining their current commitment state.

Return Value

NtFreeVirtualMemory returns either STATUS_SUCCESS or an error status code. Possible error status codes include the following.

RETURN CODE	**DESCRIPTION**
STA TUS _AC CES S_D ENI ED	A process has requested access to an object, but has not been granted those access rights.
STA TUS _IN VAL ID_ HA NDL E	An invalid *ProcessHandle* value was specified.

STA TUS _OB JEC T_T YPE _MI SM ATC H	There is a mismatch between the type of object required by the requested operation and the type of object that is specified in the request.

Remarks

Each page in the process's virtual address space is in one of the three states described in the following table.

STATE	MEANING
FREE	The page is neither committed nor reserved. The page is not accessible to the process. Attempting to read from or write to a free page results in an access violation exception. You can use **NtFreeVirtualMemory** to put reserved or committed pages into the free state.
RESERVED	The page is reserved. The range of addresses cannot be used by other allocation functions. The page is not accessible to the process and has no physical storage associated with it. Attempting to read from or write to a reserved page results in an access violation exception. You can use **NtFreeVirtualMemory** to put committed memory pages into the reserved state, and to put reserved memory pages into the free state.
COMMITTED	The page is committed. Physical storage in memory or in the paging file on disk is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read from or write to that page. When a process terminates, the system releases all storage for committed pages. You can use NtAllocateVirtualMemory to put committed memory pages into either the reserved or free state.

NtFreeVirtualMemory can perform the following operations:

- Decommit a region of committed or uncommitted pages. After this operation, the pages are in the reserved state.
- Release a region of reserved pages. After this operation, the pages are in the free state.
- Decommit and release a region of committed or uncommitted pages. After this operation, the pages are in the free state.

NtFreeVirtualMemory can decommit a range of pages that are in different states, some committed and some uncommitted. This means that you can decommit a range of pages without first determining the current commitment state of each page. Decommitting a page releases its physical storage, either in memory or in the paging file on disk.

If a page is decommitted but not released, its state changes to reserved. You can subsequently call **NtAllocateVirtualMemory** to commit it, or **NtFreeVirtualMemory** to release it. Attempting to read from or

write to a reserved page results in an access violation exception.

NtFreeVirtualMemory can release a range of pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state of each page. The entire range of pages originally reserved by NtAllocateVirtualMemory must be released at the same time.

If a page is released, its state changes to free, and it is available for subsequent allocation operations. After memory has been released or decommitted, you can never refer to the memory again. Any information that may have been in that memory is gone forever. Attempting to read from or write to a free page results in an access violation exception. If you require information, do not decommit or free memory that contains that information.

For more information about memory management support for kernel-mode drivers, see Memory Management for Windows Drivers.

> **Note** If the call to the **NtFreeVirtualMemory** function occurs in user mode, you should use the name "**NtFreeVirtualMemory**" instead of "**ZwFreeVirtualMemory**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

NtAllocateVirtualMemory

NtFsControlFile function

1/25/2019 • 4 minutes to read • Edit Online

The **NtFsControlFile** routine sends a control code directly to a specified file system or file system filter driver, causing the corresponding driver to perform the specified action.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtFsControlFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  ULONG            FsControlCode,
  PVOID            InputBuffer,
  ULONG            InputBufferLength,
  PVOID            OutputBuffer,
  ULONG            OutputBufferLength
);
```

Parameters

`FileHandle`

Handle returned by NtCreateFile or NtOpenFile for the file object representing the file or directory on which the specified action is to be performed. The file object must have been opened for asynchronous I/O if the caller specifies an *Event*, *ApcRoutine*, and an APC context (in *ApcContext*), or a completion context (in *ApcContext*).

`Event`

Handle for a caller-created event. If this parameter is supplied, the caller will be put into a wait state until the requested operation is completed and the given event is set to the Signaled state. This parameter is optional and can be **NULL**. It must be **NULL** if the caller will wait for the *FileHandle* to be set to the Signaled state.

`ApcRoutine`

Address of a caller-supplied APC routine to be called when the requested operation completes. This parameter is optional and can be **NULL**. It must be **NULL** if there is an I/O completion object associated with the file object.

`ApcContext`

Pointer to a caller-determined context area. This parameter value is used as the APC context if the caller supplies an APC, or is used as the completion context if an I/O completion object has been associated with the file object. When the operation completes, either the APC context is passed to the APC, if one was specified, or the completion context is included as part of the completion message that the I/O Manager posts to the associated I/O completion object.

This parameter is optional and can be **NULL**. It must be **NULL** if *ApcRoutine* is **NULL** and there is no I/O completion object associated with the file object.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the

operation. For successful calls that return data, the number of bytes written to the *OutputBuffer* is returned in the **Information** member of this structure.

`FsControlCode`

FSCTL_*XXX* code that indicates which file system control operation is to be carried out. The value of this parameter determines the formats and required lengths of the *InputBuffer* and *OutputBuffer*, as well as which of the following parameter pairs are required. For detailed information about the system-defined FSCTL_*XXX* codes, see the "Remarks" section of the reference entry for DeviceIoControl in the Microsoft Windows SDK documentation.

`InputBuffer`

Pointer to a caller-allocated input buffer that contains device-specific information to be given to the target driver. If *FsControlCode* specifies an operation that does not require input data, this pointer is optional and can be **NULL**.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. This value is ignored if *InputBuffer* is **NULL**.

`OutputBuffer`

Pointer to a caller-allocated output buffer in which information is returned from the target driver. If *FsControlCode* specifies an operation that does not produce output data, this pointer is optional and can be **NULL**.

`OutputBufferLength`

Size, in bytes, of the buffer at *OutputBuffer*. This value is ignored if *OutputBuffer* is **NULL**.

Return Value

NtFsControlFile returns STATUS_SUCCESS or an appropriate NTSTATUS value such as one of the following:

Remarks

NtFsControlFile provides a consistent view of the input and output data to the system and to kernel-mode drivers, while providing applications and underlying drivers with a driver-dependent method of specifying a communications interface.

If the caller opened the file for asynchronous I/O (with neither FILE_SYNCHRONOUS_*XXX* create/open option set), the specified event, if any, will be set to the signaled state when the device control operation completes. Otherwise, the file object specified by *FileHandle* will be set to the signaled state. If an *ApcRoutine* was specified, it is called with the *ApcContext* and *IoStatusBlock* pointers.

The following FSCTL codes are currently documented for kernel-mode drivers:

FSCTL_DELETE_REPARSE_POINT

FSCTL_GET_REPARSE_POINT

FSCTL_OPBATCH_ACK_CLOSE_PENDING

FSCTL_OPLOCK_BREAK_ACK_NO_2

FSCTL_OPLOCK_BREAK_ACKNOWLEDGE

FSCTL_OPLOCK_BREAK_NOTIFY

FSCTL_REQUEST_BATCH_OPLOCK

FSCTL_REQUEST_FILTER_OPLOCK

FSCTL_REQUEST_OPLOCK_LEVEL_1

FSCTL_REQUEST_OPLOCK_LEVEL_2

FSCTL_SET_REPARSE_POINT

For more information about system-defined FSCTL_*XXX* codes, see the "Remarks" section of the reference entry for DeviceIoControl in the Microsoft Windows SDK documentation.

For more information about system-defined IOCTL_*XXX* codes, and about defining driver-specific IOCTL_*XXX* or FSCTL_*XXX* values, see Using I/O Control Codes in the *Kernel Mode Architecture Guide* and *Device Input and Output Control Codes* in the Windows SDK documentation.

Minifilters should use FltFsControlFile instead of **NtFsControlFile**.

Callers of **NtFsControlFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **NtFsControlFile** function occurs in user mode, you should use the name "**NtFsControlFile**" instead of "**ZwFsControlFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltFsControlFile

IRP_MJ_FILE_SYSTEM_CONTROL

IoGetFunctionCodeFromCtlCode

IoIsOperationSynchronous

Using I/O Control Codes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateFile

ZwDeviceIoControlFile

ZwOpenFile

NtLockFile function

1/25/2019 • 2 minutes to read • Edit Online

The **NtLockFile** routine requests a byte-range lock for the specified file.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtLockFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PLARGE_INTEGER   ByteOffset,
  PLARGE_INTEGER   Length,
  ULONG            Key,
  BOOLEAN          FailImmediately,
  BOOLEAN          ExclusiveLock
);
```

Parameters

`FileHandle`

A handle for the file on which a byte-range lock is requested.

`Event`

A handle to a caller-created event. If not **NULL**, the caller is placed into a wait state until the operation succeeds, at which time the event is set into the Signaled state.

`ApcRoutine`

A pointer to a caller-supplied APC routine that is executed after the operation completes. Can be **NULL**.

`ApcContext`

A pointer to a caller-specified context for the APC routine. This value is passed to the APC routine when it is executed. Can be **NULL**.

`IoStatusBlock`

A pointer to an IO_STATUS_BLOCK structure that contains the final status.

`ByteOffset`

A pointer to a variable that specifies the starting byte offset of the range to lock.

`Length`

A pointer to a variable that specifies the length in bytes of the range to lock.

`Key`

A caller-assigned value used to describe groups of related locks. This value should be set to zero.

`FailImmediately`

If **TRUE**, immediately return if the file cannot be locked. If **FALSE**, wait for the lock request to be granted.

`ExclusiveLock`

If **TRUE**, byte-range lock is exclusive; otherwise, shared lock.

Return Value

The **NtLockFile** routine returns STATUS_SUCCESS or an appropriate error NTSTATUS value. Possible NTSTATUS values include the following:

RETURN CODE	DESCRIPTION
STA TUS _IN SUF FICI ENT _RE SO URC ES	Insufficient resources exist to grant the byte-range lock for the specified file.
STA TUS _LO CK_ NO T_G RA NTE D	The byte-range lock was not granted for the specified file.

Remarks

Callers of **NtLockFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **NtLockFile** function occurs in user mode, you should use the name "**NtLockFile**" instead of "**ZwLockFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal

Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwUnlockFile

NtOpenFile function

1/25/2019 • 2 minutes to read • Edit Online

The **NtOpenFile** routine opens an existing file, directory, device, or volume.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenFile(
  PHANDLE            FileHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PIO_STATUS_BLOCK   IoStatusBlock,
  ULONG              ShareAccess,
  ULONG              OpenOptions
);
```

Parameters

`FileHandle`

Pointer to a HANDLE variable that receives a handle to the file.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. For more information, see the *DesiredAccess* parameter of NtCreateFile.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested operation.

`ShareAccess`

Specifies the type of share access for the file. For more information, see the *ShareAccess* parameter of NtCreateFile.

`OpenOptions`

Specifies the options to apply when opening the file. For more information, see the *CreateOptions* parameter of NtCreateFile.

Return Value

NtOpenFile returns STATUS_SUCCESS or the appropriate NTSTATUS error code. In the latter case, the caller can find more information about the cause of the failure by checking the *IoStatusBlock* parameter.

Remarks

NtOpenFile supplies a handle that the caller can use to manipulate a file's data, or the file object's state and attributes. **NtOpenFile** provides a subset of the functionality provided by NtCreateFile. For more information, see Using Files in a Driver.

Once the handle pointed to by *FileHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

Callers of **NtOpenFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenFile**" instead of "**ZwOpenFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateFile

NtOpenProcessTokenEx function

1/25/2019 • 2 minutes to read • Edit Online

The **NtOpenProcessTokenEx** routine opens the access token associated with a process.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcessTokenEx(
  HANDLE      ProcessHandle,
  ACCESS_MASK DesiredAccess,
  ULONG       HandleAttributes,
  PHANDLE     TokenHandle
);
```

Parameters

`ProcessHandle`

Handle to the process whose access token is to be opened. The handle must have PROCESS_QUERY_INFORMATION access. Use the **NtCurrentProcess** macro, defined in Ntddk.h, to specify the current process.

`DesiredAccess`

ACCESS_MASK structure specifying the requested types of access to the access token. These requested access types are compared with the token's discretionary access-control list (DACL) to determine which accesses are granted or denied.

`HandleAttributes`

Attributes for the access token handle. Only OBJ_KERNEL_HANDLE is currently supported. If the caller is not running in the system process context, it must specify OBJ_KERNEL_HANDLE for this parameter.

`TokenHandle`

Pointer to a caller-allocated variable that receives a handle to the newly opened access token.

Return Value

NtOpenProcessTokenEx returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_ACCESS_DENIED	*ProcessHandle* did not have PROCESS_QUERY_INFORMATION access.

STATUS_INSUFFICIENT_RESOURCES	A new token handle could not be allocated.
STATUS_INVALID_HANDLE	*ProcessHandle* was not a valid handle.
STATUS_INVALID_PARAMETER	The specified *HandleAttributes* did not include OBJ_KERNEL_HANDLE.
STATUS_OBJECT_TYPE_MISMATCH	*ProcessHandle* was not a process handle.
STATUS_PRIVILEGE_NOT_HELD	The caller does not have the privilege (SeSecurityPrivilege) necessary to create a token handle with the access specified in the *DesiredAccess* parameter.

STATUS_QUOTA_EXCEEDED	The process's memory quota is not sufficient to allocate the token handle.
STATUS_UNSUCCESSFUL	The token handle could not be created.

Remarks

NtOpenProcessTokenEx opens the access token associated with a process and returns a handle for that token.

Any handle obtained by calling **NtOpenProcessTokenEx** must eventually be released by calling **NtClose**.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE attribute for the *HandleAttributes* parameter of **NtOpenProcessTokenEx**. This restricts the use of the handle returned by **NtOpenProcessTokenEx** to processes running in kernel mode. Otherwise, the handle can be accessed by the process in whose context the driver is running.

For more information about security and access control, see the documentation on these topics in the Windows SDK.

> **Note** If the call to the **NtOpenProcessTokenEx** function occurs in user mode, you should use the name "**NtOpenProcessTokenEx**" instead of "**ZwOpenProcessTokenEx**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

ACL

PsDereferencePrimaryToken

Using Nt and Zw Versions of the Native System Services Routines

NtClose

NtOpenThreadTokenEx

NtOpenThreadTokenEx function

1/25/2019 • 3 minutes to read • Edit Online

The **NtOpenThreadTokenEx** routine opens the access token associated with a thread.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenThreadTokenEx(
  HANDLE      ThreadHandle,
  ACCESS_MASK DesiredAccess,
  BOOLEAN     OpenAsSelf,
  ULONG       HandleAttributes,
  PHANDLE     TokenHandle
);
```

Parameters

`ThreadHandle`

Handle to the thread whose access token is to be opened. The handle must have THREAD_QUERY_INFORMATION access. Use the **NtCurrentThread** macro to specify the current thread.

`DesiredAccess`

ACCESS_MASK structure specifying the requested types of access to the access token. These requested access types are compared with the token's discretionary access-control list (DACL) to determine which access rights are granted or denied.

`OpenAsSelf`

Boolean value specifying whether the access check is to be made against the security context of the thread calling **NtOpenThreadTokenEx** or against the security context of the process for the calling thread.

If this parameter is **FALSE**, the access check is performed using the security context for the calling thread. If the thread is impersonating a client, this security context can be that of a client process. If this parameter is **TRUE**, the access check is made using the security context of the process for the calling thread.

`HandleAttributes`

Attributes for the created handle. Only OBJ_KERNEL_HANDLE is currently supported. If the caller is not running in the system process context, it must specify OBJ_KERNEL_HANDLE for this parameter.

`TokenHandle`

Pointer to a caller-allocated variable that receives a handle to the newly opened access token.

Return Value

NtOpenThreadTokenEx returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION

STATUS_ACCESS_DENIED	*ThreadHandle* did not have THREAD_QUERY_INFORMATION access.
STATUS_CANT_OPEN_ANONYMOUS	The client requested the SecurityAnonymous impersonation level. However, an anonymous token cannot be opened. For more information, see SECURITY_IMPERSONATION_LEVEL.
STATUS_INVALID_HANDLE	*ThreadHandle* was not a valid handle.
STATUS_INVALID_PARAMETER	The specified *HandleAttributes* did not include OBJ_KERNEL_HANDLE.
STATUS_NO_TOKEN	An attempt has been made to open a token associated with a thread that is not currently impersonating a client.

STA **TUS** **_OB** **JEC** **T_T** **YPE** **_MI** **SM** **ATC** **H**	*ThreadHandle* was not a thread handle.

Remarks

NtOpenThreadTokenEx opens the access token associated with a thread and returns a handle for that token.

The *OpenAsSelf* parameter allows a server process to open the access token for a client process when the client process has specified the SecurityIdentification impersonation level for the **SECURITY_IMPERSONATION_LEVEL** enumerated type. Without this parameter, the calling process is not be able to open the client's access token using the client's security context because it is impossible to open executive-level objects using the SecurityIdentification impersonation level.

Any handle obtained by calling **NtOpenThreadTokenEx** must eventually be released by calling **NtClose**.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE attribute for the *HandleAttributes* parameter of **NtOpenThreadTokenEx**. This restricts the use of the handle returned by **NtOpenThreadTokenEx** to processes running in kernel mode. Otherwise, the handle can be accessed by the process in whose context the driver is running.

For more information about security and access control, see the documentation on these topics in the Windows SDK.

> **Note** If the call to the **NtOpenThreadTokenEx** function occurs in user mode, you should use the name "**NtOpenThreadTokenEx**" instead of "**ZwOpenThreadTokenEx**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

ACL

PsDereferencePrimaryToken

SECURITY_IMPERSONATION_LEVEL

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwOpenProcessTokenEx

NtQueryDirectoryFile function

1/25/2019 • 6 minutes to read • Edit Online

The **NtQueryDirectoryFile** routine returns various kinds of information about files in the directory specified by a given file handle.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryDirectoryFile(
  HANDLE                 FileHandle,
  HANDLE                 Event,
  PIO_APC_ROUTINE        ApcRoutine,
  PVOID                  ApcContext,
  PIO_STATUS_BLOCK       IoStatusBlock,
  PVOID                  FileInformation,
  ULONG                  Length,
  FILE_INFORMATION_CLASS FileInformationClass,
  BOOLEAN                ReturnSingleEntry,
  PUNICODE_STRING        FileName,
  BOOLEAN                RestartScan
);
```

Parameters

`FileHandle`

A handle returned by NtCreateFile or NtOpenFile for the file object that represents the directory for which information is being requested. The file object must have been opened for asynchronous I/O if the caller specifies a non-**NULL** value for *Event* or *ApcRoutine*.

`Event`

An optional handle for a caller-created event. If this parameter is supplied, the caller will be put into a wait state until the requested operation is completed and the given event is set to the Signaled state. This parameter is optional and can be **NULL**. It must be **NULL** if the caller will wait for the *FileHandle* to be set to the Signaled state.

`ApcRoutine`

An address of an optional, caller-supplied APC routine to be called when the requested operation completes. This parameter is optional and can be **NULL**. If there is an I/O completion object associated with the file object, this parameter must be **NULL**.

`ApcContext`

An optional pointer to a caller-determined context area if the caller supplies an APC or if an I/O completion object is associated with the file object. When the operation completes, this context is passed to the APC, if one was specified, or is included as part of the completion message that the I/O Manager posts to the associated I/O completion object.

This parameter is optional and can be **NULL**. It must be **NULL** if *ApcRoutine* is **NULL** and there is no I/O completion object associated with the file object.

`IoStatusBlock`

A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the operation. For successful calls that return data, the number of bytes written to the *FileInformation* buffer is returned in the structure's **Information** member.

`FileInformation`

A pointer to a buffer that receives the desired information about the file. The structure of the information returned in the buffer is defined by the *FileInformationClass* parameter.

`Length`

The size, in bytes, of the buffer pointed to by *FileInformation*. The caller should set this parameter according to the given *FileInformationClass*.

`FileInformationClass`

The type of information to be returned about files in the directory. One of the following.

VALUE	MEANING
FileBothDirectoryInformation	Return a FILE_BOTH_DIR_INFORMATION structure for each file.
FileDirectoryInformation	Return a FILE_DIRECTORY_INFORMATION structure for each file.
FileFullDirectoryInformation	Return a FILE_FULL_DIR_INFORMATION structure for each file.
FileIdBothDirectoryInformation	Return a FILE_ID_BOTH_DIR_INFORMATION structure for each file.
FileIdFullDirectoryInformation	Return a FILE_ID_FULL_DIR_INFORMATION structure for each file.
FileNamesInformation	Return a FILE_NAMES_INFORMATION structure for each file.
FileObjectIdInformation	Return a FILE_OBJECTID_INFORMATION structure for each file. This information class is valid only for NTFS volumes on Windows 2000 and later versions of Windows.
FileReparsePointInformation	Return a single FILE_REPARSE_POINT_INFORMATION structure for the directory.

`ReturnSingleEntry`

Set to **TRUE** if only a single entry should be returned, **FALSE** otherwise. If this parameter is **TRUE**, **NtQueryDirectoryFile** returns only the first entry that is found.

`FileName`

An optional pointer to a caller-allocated Unicode string containing the name of a file (or multiple files, if wildcards are used) within the directory specified by *FileHandle*. This parameter is optional and can be **NULL**.

If *FileName* is not **NULL**, only files whose names match the *FileName* string are included in the directory scan. If *FileName* is **NULL**, all files are included.

The *FileName* is used as a search expression and is captured on the very first call to **NtQueryDirectoryFile** for a given handle. Subsequent calls to **NtQueryDirectoryFile** will use the search expression set in the first call. The *FileName* parameter passed to subsequent calls will be ignored.

`RestartScan`

Set to **TRUE** if the scan is to start at the first entry in the directory. Set to **FALSE** if resuming the scan from a previous call.

When the **NtQueryDirectoryFile** routine is called for a particular handle, the *RestartScan* parameter is treated as if it were set to **TRUE**, regardless of its value. On subsequent **NtQueryDirectoryFile** calls, the value of the *RestartScan* parameter is honored.

Return Value

The **NtQueryDirectoryFile**routine returns STATUS_SUCCESS or an appropriate error status. Note that the set of error status values that can be returned is file-system-specific. **NtQueryDirectoryFile**also returns the number of bytes actually written to the given *FileInformation* buffer in the **Information** member of *IoStatusBlock*.

Remarks

The **NtQueryDirectoryFile** routine returns information about files that are contained in the directory represented by *FileHandle*.

If provided, the value of the *FileName* parameter determines the entries that are included in the directory scan for all subsequent calls to **NtQueryDirectoryFile** for a given *FileHandle*.

If there is at least one matching entry, **NtQueryDirectoryFile** creates a **FILE_*XXX*_INFORMATION** structure for each entry and stores them in the buffer.

Assuming that at least one matching directory entry is found, the number of entries for which information is returned is the *smallest* of the following:

- One entry, if *ReturnSingleEntry* is **TRUE** and *FileName* is **NULL**.
- The number of entries that match the *FileName* string, if *FileName* is not **NULL**. (Note that if the string contains no wildcards, there can be at most one matching entry.)
- The number of entries whose information fits into the specified buffer.
- The number of entries contained in the directory.

On the first call to **NtQueryDirectoryFile**, if the structure created for the first entry found is too large to fit into the output buffer, the routine writes the fixed portion of the structure to the output buffer. The routine then writes to the output buffer as much of the *FileName* string as will fit. (The fixed portion of the structure consists of all fields except the final *FileName* string. On the first call, but not on subsequent calls, the I/O system ensures that the buffer is large enough to hold the fixed portion of the appropriate **FILE_*XXX*_INFORMATION** structure.) When this happens, **NtQueryDirectoryFile** returns an appropriate status value such as STATUS_BUFFER_OVERFLOW.

On each call, **NtQueryDirectoryFile** returns as many **FILE_*XXX*_INFORMATION** structures (one per directory entry) as can be contained entirely in the buffer pointed to by *FileInformation*. On the first call, **NtQueryDirectoryFile** returns STATUS_SUCCESS only if the output buffer contains at least one complete structure. On subsequent calls, if the output buffer contains no structures, **NtQueryDirectoryFile** returns STATUS_SUCCESS but sets *IoStatusBlock*->**Information** = 0 to notify the caller of this condition. In this case, the caller should allocate a larger buffer and call **NtQueryDirectoryFile** again. No information about any remaining entries is reported. Thus, except in the cases listed above where only one entry is returned, **NtQueryDirectoryFile** must be called at least twice to enumerate the contents of an entire directory.

When calling **NtQueryDirectoryFile**, you may see changes made to the directory that occur in parallel with **NtQueryDirectoryFile** calls. This behavior is dependent on the implementation of the underlying file system.

The final call to **NtQueryDirectoryFile** returns an empty output buffer and reports an appropriate status value

such as STATUS_NO_MORE_FILES.

If **NtQueryDirectoryFile** is called multiple times on the same directory and some other operation changes the contents of that directory, any changes may or may not be seen, depending on the timing of the operations.

NtQueryDirectoryFilereturns zero in any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by the file system.

Callers of **NtQueryDirectoryFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

For information about other file information query routines, see File Objects.

> **Note** If the call to the **NtQueryDirectoryFile** function occurs in user mode, you should use the name "**NtQueryDirectoryFile**" instead of "**ZwQueryDirectoryFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_BOTH_DIR_INFORMATION

FILE_DIRECTORY_INFORMATION

FILE_FULL_DIR_INFORMATION

FILE_ID_BOTH_DIR_INFORMATION

FILE_ID_FULL_DIR_INFORMATION

FILE_NAMES_INFORMATION

FILE_OBJECTID_INFORMATION

FILE_REPARSE_POINT_INFORMATION

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

NtCreateFile

NtOpenFile

NtQueryInformationFile function

1/25/2019 • 5 minutes to read • Edit Online

The **NtQueryInformationFile** routine returns various kinds of information about a file object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationFile(
  HANDLE                 FileHandle,
  PIO_STATUS_BLOCK       IoStatusBlock,
  PVOID                  FileInformation,
  ULONG                  Length,
  FILE_INFORMATION_CLASS FileInformationClass
);
```

Parameters

`FileHandle`

Handle to a file object. The handle is created by a successful call to NtCreateFile or NtOpenFile.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the operation. The **Information** member receives the number of bytes that this routine actually writes to the *FileInformation* buffer.

`FileInformation`

Pointer to a caller-allocated buffer into which the routine writes the requested information about the file object. The *FileInformationClass* parameter specifies the type of information that the caller requests.

`Length`

The size, in bytes, of the buffer pointed to by *FileInformation*.

`FileInformationClass`

Specifies the type of information to be returned about the file, in the buffer that *FileInformation* points to. Device and intermediate drivers can specify any of the following FILE_INFORMATION_CLASS values.

FILE_INFORMATION_CLASS VALUE	TYPE OF INFORMATION RETURNED
FileAccessInformation	A FILE_ACCESS_INFORMATION structure. This structure contains an access mask. For more information about access masks, see ACCESS_MASK.
FileAlignmentInformation	A FILE_ALIGNMENT_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*. This information is useful if the file was opened with the FILE_NO_INTERMEDIATE_BUFFERING flag specified in the *CreateOptions* parameter.

FileAllInformation	A FILE_ALL_INFORMATION structure. By combining several file-information structures into a single structure, **FILE_ALL_INFORMATION** reduces the number of queries required to obtain information about a file.
FileAttributeTagInformation	A FILE_ATTRIBUTE_TAG_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.
FileBasicInformation	A FILE_BASIC_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.
FileCaseSensitiveInformation	A FILE_CASE_SENSITIVE_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter. This value is available starting with Windows 10, version 1803.
FileCaseSensitiveInformationForceAccessCheck	A FILE_CASE_SENSITIVE_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter. This is a special version of the FileCaseSensitiveInformation operation that is used to force the IOManager to perform access checks for the kernel-mode driver, similar to the checks that apply to a user-mode caller. This operation is only recognized by the IOManager and a file system should never receive it. This value is available starting with Windows 10, version 1803.
FileEaInformation	A FILE_EA_INFORMATION structure. This structure specifies the size of the extended attributes block that is associated with the file.
FileInternalInformation	A FILE_INTERNAL_INFORMATION structure. This structure specifies a 64-bit file ID that uniquely identifies a file in NTFS. On other file systems, this file ID is not guaranteed to be unique.
FileIoPriorityHintInformation	A FILE_IO_PRIORITY_HINT_INFORMATION structure. The caller must have opened the file with the FILE_READ_DATA flag specified in the *DesiredAccess* parameter.
FileModeInformation	A FILE_MODE_INFORMATION structure. This structure contains a set of flags that specify the mode in which the file can be accessed. These flags are a subset of the options that can be specified in the *CreateOptions* parameter of the IoCreateFile routine.
FileNameInformation	A FILE_NAME_INFORMATION structure. The structure can contain the file's full path or only a portion of it. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*. For more information about the file-name syntax, see the Remarks section later in this topic.
FileNetworkOpenInformation	A FILE_NETWORK_OPEN_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.

FilePositionInformation	A FILE_POSITION_INFORMATION structure. The caller must have opened the file with the *DesiredAccess* FILE_READ_DATA or FILE_WRITE_DATA flag specified in the *DesiredAccess* parameter, and with the FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT flag specified in the *CreateOptions*parameter.
FileStandardInformation	A FILE_STANDARD_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*.
FileIsRemoteDeviceInformation	A FILE_IS_REMOTE_DEVICE_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*.

Return Value

NtQueryInformationFile returns STATUS_SUCCESS or an appropriate NTSTATUS error code.

Remarks

NtQueryInformationFile returns information about the specified file object. Note that it returns zero in any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by a particular device or file system.

When *FileInformationClass* = **FileNameInformation**, the file name is returned in the FILE_NAME_INFORMATION structure. The precise syntax of the file name depends on a number of factors:

- If you opened the file by submitting a full path to NtCreateFile, **NtQueryInformationFile** returns that full path.
- If the **ObjectAttributes->RootDirectory** handle was opened by name in a call to **NtCreateFile**, and subsequently the file was opened by **NtCreateFile** relative to this root-directory handle, **NtQueryInformationFile** returns the full path.
- If the **ObjectAttributes->RootDirectory** handle was opened by file ID (using the FILE_OPEN_BY_FILE_ID flag) in a call to **NtCreateFile**, and subsequently the file was opened by **NtCreateFile** relative to this root-directory handle, **NtQueryInformationFile** returns the relative path.
- However, if the user has **SeChangeNotifyPrivilege** (described in the Microsoft Windows SDK documentation), **NtQueryInformationFile** returns the full path in all cases.
- If only the relative path is returned, the file name string will not begin with a backslash.
- If the full path and file name are returned, the string will begin with a single backslash, regardless of its location. Thus the file C:\dir1\dir2\filename.ext will appear as \dir1\dir2\filename.ext, while the file \server\share\dir1\dir2\filename.ext will appear as \server\share\dir1\dir2\filename.ext.

If **NtQueryInformationFile** fails because of a buffer overflow, drivers that implement **FileNameInformation** should return as many WCHAR characters of the file name as will fit in the buffer and specify the full length that is required in the *FileNameLength* parameter of the FILE_NAME_INFORMATION structure. You should reissue the query by using the file name length so that you can retrieve the full file name. Drivers that do not follow this pattern might require a gradual increase in length until they retrieve the full file name. For more information about working with files, see Using Files in a Driver.

Callers of **NtQueryInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

Note If the call to this function occurs in user mode, you should use the name "**NtQueryInformationFile**" instead of "**ZwQueryInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_ACCESS_INFORMATION

FILE_ALIGNMENT_INFORMATION

FILE_ALL_INFORMATION

FILE_ATTRIBUTE_TAG_INFORMATION

FILE_BASIC_INFORMATION

FILE_EA_INFORMATION

FILE_INTERNAL_INFORMATION

FILE_IO_PRIORITY_HINT_INFORMATION

FILE_IS_REMOTE_DEVICE_INFORMATION

FILE_MODE_INFORMATION

FILE_NAME_INFORMATION

FILE_NETWORK_OPEN_INFORMATION

FILE_POSITION_INFORMATION

FILE_STANDARD_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

NtCreateFile

NtSetInformationFile

NtQueryInformationToken function

1/25/2019 • 3 minutes to read • Edit Online

The **NtQueryInformationToken** routine retrieves a specified type of information about an access token. The calling process must have appropriate access rights to obtain the information.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationToken(
  HANDLE                  TokenHandle,
  TOKEN_INFORMATION_CLASS TokenInformationClass,
  PVOID                   TokenInformation,
  ULONG                   TokenInformationLength,
  PULONG                  ReturnLength
);
```

Parameters

`TokenHandle`

Handle for an access token from which information is to be retrieved. If *TokenInformationClass* is set to **TokenSource**, the handle must have TOKEN_QUERY_SOURCE access. For all other *TokenInformationClass* values, the handle must have TOKEN_QUERY access. For more information about access rights for access-token objects, see the Security section of the Windows SDK documentation.

`TokenInformationClass`

A value from the TOKEN_INFORMATION_CLASS enumerated type identifying the type of information to be retrieved. The possible values for this parameter are listed in the *TokenInformationClass* Value column of the table shown in the description of the *TokenInformation* parameter.

`TokenInformation`

Pointer to a caller-allocated buffer that receives the requested information about the token. The structure put into this buffer depends upon the value of *TokenInformationClass*, as shown in the following table. All structures must be aligned on a 32-bit boundary.

***TOKENINFORMATIONCLASS* VALUE**	**EFFECT ON *TOKENINFORMATION* BUFFER**
TokenDefaultDacl	The buffer receives a TOKEN_DEFAULT_DACL structure containing the default DACL for newly created objects.
TokenGroups	The buffer receives a TOKEN_GROUPS structure containing the group accounts associated with the token.
TokenImpersonationLevel	The buffer receives a SECURITY_IMPERSONATION_LEVEL value indicating the impersonation level of the token. If the access token is not an impersonation token, the call to **NtQueryInformationToken** fails.
TokenOwner	The buffer receives a TOKEN_OWNER structure containing the default owner SID for newly created objects.

TokenPrimaryGroup	The buffer receives a TOKEN_PRIMARY_GROUP structure containing the default primary group SID for newly created objects.
TokenPrivileges	The buffer receives a TOKEN_PRIVILEGES structure containing the token's privileges.
TokenSessionId	The buffer receives a 32-bit value specifying the Terminal Services session identifier associated with the token. If the token is associated with the Terminal Server console session, the session identifier is zero. A nonzero session identifier indicates a Terminal Services client session. In a non-Terminal Services environment, the session identifier is zero.
TokenSource	The buffer receives a TOKEN_SOURCE structure containing the source of the token. TOKEN_QUERY_SOURCE access is needed to retrieve this information.
TokenStatistics	The buffer receives a TOKEN_STATISTICS structure containing various token statistics.
TokenType	The buffer receives a TOKEN_TYPE value indicating whether the token is a primary or impersonation token.
TokenUser	The buffer receives a TOKEN_USER structure containing the token's user account.

`TokenInformationLength`

Length, in bytes, of the caller-allocated *TokenInformation* buffer.

`ReturnLength`

Pointer to a caller-allocated variable that receives the actual length, in bytes, of the information returned in the *TokenInformation* buffer. If either of the following conditions is true, no data is returned in the *TokenInformation* buffer:

- The size of the requested token information structure is greater than *TokenInformationLength*. In this case, *ReturnLength* receives the actual number of bytes needed to store the requested information.
- The value of *TokenInformationClass* is **TokenDefaultDacl**, and there is no default DACL established for the token. In this case, *ReturnLength* receives zero.

Return Value

NtQueryInformationToken returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STA TUS _AC CES S_D ENI ED	*TokenHandle* did not have the required access.

STATUS_BUFFER_TOO_SMALL	The size of the requested token information structure is greater than *TokenInformationLength*. The number of bytes required is returned in *ReturnLength*.
STATUS_INVALID_HANDLE	*TokenHandle* was not a valid handle.
STATUS_INVALID_INFO_CLASS	*TokenInformationClass* was not a valid token information class.
STATUS_OBJECT_TYPE_MISMATCH	*TokenHandle* was not a token handle.

Remarks

The **NtQueryInformationToken** routine can be used by a file system or file system filter driver to determine the SID of the caller that initiated the request during IRP_MJ_CREATE processing. If **TokenUser** is specified for the *TokenInformationClass* parameter passed to **NtQueryInformationToken**, a TOKEN_USER structure is returned in the buffer pointed to by the *TokenInformation* parameter. This returned buffer contains an SID_AND_ATTRIBUTES structure with the user **SID**.

For more information about security and access control, see the documentation on these topics in the Windows SDK.

Note If the call to the **NtQueryInformationToken** function occurs in user mode, you should use the name "**NtQueryInformationToken**" instead of "**ZwQueryInformationToken**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACL

IRP_MJ_CREATE

SECURITY_IMPERSONATION_LEVEL

SID

SID_AND_ATTRIBUTES

SeQueryInformationToken

TOKEN_DEFAULT_DACL

TOKEN_GROUPS

TOKEN_INFORMATION_CLASS

TOKEN_OWNER

TOKEN_PRIMARY_GROUP

TOKEN_PRIVILEGES

TOKEN_SOURCE

TOKEN_STATISTICS

TOKEN_TYPE

TOKEN_USER

Using Nt and Zw Versions of the Native System Services Routines

ZwSetInformationToken

NtQueryObject function

1/25/2019 • 2 minutes to read • Edit Online

The **NtQueryObject** routine provides information about a supplied object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryObject(
  HANDLE                   Handle,
  OBJECT_INFORMATION_CLASS ObjectInformationClass,
  PVOID                    ObjectInformation,
  ULONG                    ObjectInformationLength,
  PULONG                   ReturnLength
);
```

Parameters

`Handle`

A handle to the object to obtain information about.

`ObjectInformationClass`

Specifies an OBJECT_INFORMATION_CLASS value that determines the type of information returned in the *ObjectInformation* buffer.

`ObjectInformation`

A pointer to a caller-allocated buffer that receives the requested information.

`ObjectInformationLength`

Specifies the size, in bytes, of the *ObjectInformation* buffer.

`ReturnLength`

A pointer to a variable that receives the size, in bytes, of the requested key information. If **NtQueryObject** returns STATUS_SUCCESS, the variable contains the amount of data returned. If **NtQueryObject** returns STATUS_BUFFER_OVERFLOW or STATUS_BUFFER_TOO_SMALL, you can use the value of the variable to determine the required buffer size.

Return Value

NtQueryObject returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION

STA TUS _AC CES S_D ENI ED	There were insufficient permissions to perform this query.
STA TUS _IN VAL ID_ HA NDL E	The supplied object handle is invalid.
STA TUS _INF O_L ENG TH_ MIS MA TCH	The info length is not sufficient to hold the data.

Remarks

If the call to the **NtQueryObject** function occurs in user mode, you should use the name "**NtQueryObject**" instead of "**ZwQueryObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

OBJECT_INFORMATION_CLASS

PUBLIC_OBJECT_BASIC_INFORMATION

PUBLIC_OBJECT_TYPE_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

NtQueryQuotaInformationFile function

1/25/2019 • 3 minutes to read • Edit Online

The **NtQueryQuotaInformationFile** routine retrieves quota entries associated with the volume specified by the *FileHandle* parameter.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryQuotaInformationFile(
  HANDLE           FileHandle,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  BOOLEAN          ReturnSingleEntry,
  PVOID            SidList,
  ULONG            SidListLength,
  PSID             StartSid,
  BOOLEAN          RestartScan
);
```

Parameters

`FileHandle`

A handle for the file object that represents the file or volume for which the quota information is requested.

`IoStatusBlock`

The address of the caller's I/O status block.

`Buffer`

A buffer to receive the quota information for the volume. The quota information is formatted as one or more FILE_QUOTA_INFORMATION structures. The **NextEntryOffset** field in the **FILE_QUOTA_INFORMATION** structure contains the offset, in bytes, of the next quota entry in the list. If there are no more entries after the current one, this member is zero.

`Length`

The length in bytes of the buffer.

`ReturnSingleEntry`

A Boolean value that indicates if only a single entry should be returned rather than filling the buffer with as many entries as possible.

`SidList`

An optional list of SIDs whose quota information is to be returned. Each entry in the list is a FILE_GET_QUOTA_INFORMATION structure. The **NextEntryOffset** field in the **FILE_GET_QUOTA_INFORMATION** structure contains the offset, in bytes, of the next quota entry in the list. If there are no more entries after the current one, this member is zero.

`SidListLength`

The length in bytes of the **SID** list, if one was specified.

`StartSid`

An optional pointer to the **SID** of the entry at which to begin scanning the quota information. This parameter should be set if the returned information is to start with an entry other than the first SID. This parameter is ignored if a *SidList* parameter is specified.

`RestartScan`

A Boolean value that indicates whether the scan of the quota information is to be restarted from the beginning. Set this parameter to **TRUE** if the scan of the quota information is to start at the first entry in the volume's quota information list. Set to **FALSE** if resuming the scan from a previous call to **NtQueryQuotaInformationFile**. The caller must set this parameter to **TRUE** when calling **NtQueryQuotaInformationFile** for the first time.

Return Value

The **NtQueryQuotaInformationFile** routine returns STATUS_SUCCESS if at least one FILE_QUOTA_INFORMATION structure is returned in the *Buffer* parameter or an appropriate NTSTATUS value such as one of the following:

RETURN CODE	DESCRIPTION
STATUS_INSUFFICIENT_RESOURCES	There were insufficient resources to complete the operation. This is an error code.
STATUS_INVALID_DEVICE_REQUEST	Quotas are not enabled on the volume. This is an error code.
STATUS_INVALID_SID	The*StartSid* parameter did not contain a valid **SID**. This is an error code.

STA **TUS** **_QU** **OT** **A_LI** **ST_I** **NC** **ON** **SIST** **ENT**	The*SidList* parameter did not contain a valid, properly formed list. This is an error code.

Remarks

The amount of information returned by **NtQueryQuotaInformationFile** is based on the size of the quota information associated with the volume, the size of the buffer, and whether a specific set of entries has been requested.

A call to **NtQueryQuotaInformationFile** will result in an IRP_MJ_QUERY_QUOTA request being sent to the device object that is associated with the file object whose handle is stored in the *FileHandle* parameter.

If the underlying file system does not support quota information (FAT and CDFS file systems, for example), **NtQueryQuotaInformationFile** will fail returning STATUS_INVALID_DEVICE_REQUEST.

> **Note** If the call to the **NtQueryQuotaInformationFile** function occurs in user mode, you should use the name "**NtQueryQuotaInformationFile**" instead of "**ZwQueryQuotaInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows operating systems.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_GET_QUOTA_INFORMATION

FILE_QUOTA_INFORMATION

IO_STATUS_BLOCK

IRP_MJ_QUERY_QUOTA

IRP_MJ_SET_QUOTA

IoCheckQuotaBufferValidity

Using Nt and Zw Versions of the Native System Services Routines

ZwSetQuotaInformationFile

NtQuerySecurityObject function

1/25/2019 • 2 minutes to read • Edit Online

The **NtQuerySecurityObject** routine retrieves a copy of an object's security descriptor.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQuerySecurityObject(
  HANDLE               Handle,
  SECURITY_INFORMATION SecurityInformation,
  PSECURITY_DESCRIPTOR SecurityDescriptor,
  ULONG                Length,
  PULONG               LengthNeeded
);
```

Parameters

`Handle`

Handle for the object whose security descriptor is to be queried. This handle must have the access specified in the Meaning column of the table shown in the description of the *SecurityInformation* parameter.

`SecurityInformation`

Pointer to a SECURITY_INFORMATION value specifying the information to be queried.

VALUE	MEANING
DACL_SECURITY_INFORMATION	Indicates the discretionary access control list (DACL) of the object is being queried. Requires READ_CONTROL access.
GROUP_SECURITY_INFORMATION	Indicates the primary group identifier of the object is being queried. Requires READ_CONTROL access.
OWNER_SECURITY_INFORMATION	Indicates the owner identifier of the object is being queried. Requires READ_CONTROL access.
SACL_SECURITY_INFORMATION	Indicates the system ACL (SACL) of the object is being queried. Requires ACCESS_SYSTEM_SECURITY access.

`SecurityDescriptor`

Caller-allocated buffer that **NtQuerySecurityObject** fills with a copy of the specified security descriptor. The SECURITY_DESCRIPTOR structure is returned in self-relative format.

`Length`

Size, in bytes, of the buffer pointed to by *SecurityDescriptor*.

`LengthNeeded`

Pointer to a caller-allocated variable that receives the number of bytes required to store the copied security descriptor.

Return Value

NtQuerySecurityObject returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_ACCESS_DENIED	*Handle* did not have the required access.
STATUS_BUFFER_TOO_SMALL	The buffer is too small for the security descriptor. None of the security information was copied to the buffer.
STATUS_INVALID_HANDLE	*Handle* was not a valid handle.
STATUS_OBJECT_TYPE_MISMATCH	*Handle* was not a handle of the expected type.

Remarks

A security descriptor can be in absolute or self-relative form. In self-relative form, all members of the structure are located contiguously in memory. In absolute form, the structure only contains pointers to the members.

The NTFS file system imposes a 64K limit on the size of the security descriptor that is written to disk for a file. (The FAT file system does not support security descriptors for files.) Thus a 64K *SecurityDescriptor* buffer is guaranteed to be large enough to hold the returned **SECURITY_DESCRIPTOR** structure.

For more information about security and access control, see the documentation on these topics in the Windows SDK.

Minifilters should call FltQuerySecurityObject instead of **NtQuerySecurityObject**.

> **Note** If the call to the **NtQuerySecurityObject** function occurs in user mode, you should use the name "**NtQuerySecurityObject**" instead of "**ZwQuerySecurityObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltQuerySecurityObject

SECURITY_DESCRIPTOR

SECURITY_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwSetSecurityObject

NtQueryVirtualMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **NtQueryVirtualMemory** routine determines the state, protection, and type of a region of pages within the virtual address space of the subject process.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryVirtualMemory(
  HANDLE                   ProcessHandle,
  PVOID                    BaseAddress,
  MEMORY_INFORMATION_CLASS MemoryInformationClass,
  PVOID                    MemoryInformation,
  SIZE_T                   MemoryInformationLength,
  PSIZE_T                  ReturnLength
);
```

Parameters

`ProcessHandle`

A handle for the process in whose context the pages to be queried reside. Use the NtCurrentProcess macro to specify the current process.

`BaseAddress`

The base address of the region of pages to be queried. This value is rounded down to the next host-page- address boundary.

`MemoryInformationClass`

The memory information class about which to retrieve information. Currently, the only supported MEMORY_INFORMATION_CLASS value is **MemoryBasicInformation**.

`MemoryInformation`

A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified information class specified in the *MemoryInformationClass* parameter. When the value **MemoryBasicInformation** is passed to *MemoryInformationClass*, the *MemoryInformationClass* parameter value is a MEMORY_BASIC_INFORMATION.

`MemoryInformationLength`

Specifies the length in bytes of the memory information buffer.

`ReturnLength`

An optional pointer which, if specified, receives the number of bytes placed in the memory information buffer.

Return Value

Returns STATUS_SUCCESS if the call is successful. If the call fails, possible error codes include the following:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The specified base address is outside the range of accessible addresses.
STATUS_ACCESS_DENIED	The caller had insufficient access rights to perform the requested action.
STATUS_INFO_LENGTH_MISMATCH	The *MemoryInformation* buffer is larger than *MemoryInformationLength*.
STATUS_INVALID_INFO_CLASS	A value other than **MemoryBasicInformation** was passed to the *MemoryInformationClass* parameter.

Remarks

ZwQueryVirtualMemory determines the state of the first page within the region and then scans subsequent entries in the process address map from the base address upward until either the entire range of pages has been scanned or until a page with a non-matching set of attributes is encountered. The region attributes, the length of the region of pages with matching attributes, and an appropriate status value are returned.

If the entire region of pages does not have a matching set of attributes, then the *ReturnLength* parameter value can be used to compute the address and length of the region of pages that was not scanned.

NtQueryVirtualMemory and **ZwQueryVirtualMemory** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native

System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	ntifs.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

See Also

POWER_PLATFORM_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

NtQueryVolumeInformationFile function

1/25/2019 • 3 minutes to read • Edit Online

The **NtQueryVolumeInformationFile** routine retrieves information about the volume associated with a given file, directory, storage device, or volume.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryVolumeInformationFile(
  HANDLE               FileHandle,
  PIO_STATUS_BLOCK     IoStatusBlock,
  PVOID                FsInformation,
  ULONG                Length,
  FS_INFORMATION_CLASS FsInformationClass
);
```

Parameters

`FileHandle`

A handle to a file object returned by NtCreateFile or NtOpenFile for an open file, directory, storage device, or volume for which volume information is being requested.

`IoStatusBlock`

A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the query operation. For successful calls that return data, the number of bytes written to the *FsInformation* buffer is returned in the structure's **Information** member.

`FsInformation`

A pointer to a caller-allocated buffer that receives the desired information about the volume. The structure of the information returned in the buffer is defined by the *FsInformationClass* parameter.

`Length`

Size in bytes of the buffer pointed to by *FsInformation*. The caller should set this parameter according to the given *FsInformationClass*.

`FsInformationClass`

Type of information to be returned about the volume. Set this member to one of the following FS_INFORMATION_CLASS enumeration values.

VALUE	MEANING
FileFsAttributeInformation	Return a FILE_FS_ATTRIBUTE_INFORMATION structure containing attribute information about the file system responsible for the volume.
FileFsControlInformation	Return a FILE_FS_CONTROL_INFORMATION structure containing file system control information about the volume.

FileFsDeviceInformation	Return a FILE_FS_DEVICE_INFORMATION structure containing device information for the volume.
FileFsDriverPathInformation	Return a FILE_FS_DRIVER_PATH_INFORMATION structure containing information about whether a specified driver is in the I/O path for the volume. The caller must store the name of the driver into the **FILE_FS_DRIVER_PATH_INFORMATION** structure before calling **NtQueryVolumeInformationFile**.
FileFsFullSizeInformation	Return a FILE_FS_FULL_SIZE_INFORMATION structure containing information about the total amount of space available on the volume.
FileFsObjectIdInformation	Return a FILE_FS_OBJECTID_INFORMATION structure containing file system-specific object ID information for the volume. Note that this is not the same as the (GUID-based) unique volume name assigned by the operating system.
FileFsSizeInformation	Return a FILE_FS_SIZE_INFORMATION structure containing information about the amount of space on the volume that is available to the user associated with the calling thread.
FileFsVolumeInformation	Return a FILE_FS_VOLUME_INFORMATION containing information about the volume such as the volume label, serial number, and creation time.
FileFsSectorSizeInformation	Return a FILE_FS_SECTOR_SIZE_INFORMATION structure that contains information about the physical and logical sector sizes of a volume.

Return Value

NtQueryVolumeInformationFile returns STATUS_SUCCESS or an appropriate error status.

Remarks

NtQueryVolumeInformationFile retrieves information about the volume associated with a given file, directory, storage device, or volume.

If the *FileHandle* represents a direct device open, only *FileFsDeviceInformation* can be specified as the value of *FsInformationClass*.

NtQueryVolumeInformationFile returns zero in any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by the file system.

For information about other file information query routines, see File Objects.

Minifilters should use FltQueryVolumeInformationFile instead of **NtQueryVolumeInformationFile**.

Callers of **NtQueryVolumeInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **NtQueryVolumeInformationFile** function occurs in user mode, you should use the name "**NtQueryVolumeInformationFile**" instead of "**ZwQueryVolumeInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_FS_ATTRIBUTE_INFORMATION

FILE_FS_CONTROL_INFORMATION

FILE_FS_DEVICE_INFORMATION

FILE_FS_DRIVER_PATH_INFORMATION

FILE_FS_FULL_SIZE_INFORMATION

FILE_FS_OBJECTID_INFORMATION

FILE_FS_SIZE_INFORMATION

FILE_FS_VOLUME_INFORMATION

FltQueryVolumeInformationFile

IRP_MJ_QUERY_VOLUME_INFORMATION

IRP_MJ_SET_VOLUME_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateFile

ZwOpenFile

ZwQueryDirectoryFile

ZwQueryInformationFile

ZwSetInformationFile

ZwSetVolumeInformationFile

NtReadFile function

1/25/2019 • 4 minutes to read • Edit Online

The **NtReadFile** routine reads data from an open file.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtReadFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to NtCreateFile or NtOpenFile.

`Event`

Optionally, a handle to an event object to set to the signaled state after the read operation completes. Device and intermediate drivers should set this parameter to **NULL**.

`ApcRoutine`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`ApcContext`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested read operation. The **Information** member receives the number of bytes actually read from the file.

`Buffer`

Pointer to a caller-allocated buffer that receives the data read from the file.

`Length`

The size, in bytes, of the buffer pointed to by *Buffer*.

`ByteOffset`

Pointer to a variable that specifies the starting byte offset in the file where the read operation will begin. If an attempt is made to read beyond the end of the file, **NtReadFile** returns an error.

If the call to NtCreateFile set either of the **CreateOptions** flags FILE_SYNCHRONOUS_IO_ALERT or

FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **NtReadFile** can specify that the current file position offset be used instead of an explicit **ByteOffset** value. This specification can be made by using one of the following methods:

- Specify a pointer to a LARGE_INTEGER value with the **HighPart** member set to -1 and the **LowPart** member set to the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a **NULL** pointer for *ByteOffset*.

NtReadFile updates the current file position by adding the number of bytes read when it completes the read operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **NtReadFile**. Doing this automatically changes the current file position to that *ByteOffset* value, performs the read operation, and then updates the position according to the number of bytes actually read. This technique gives the caller atomic seek-and-read service.

`Key`

Device and intermediate drivers should set this pointer to **NULL**.

Return Value

NtReadFile returns either STATUS_SUCCESS or the appropriate NTSTATUS error code.

Remarks

Callers of **NtReadFile** must have already called NtCreateFile with the FILE_READ_DATA or GENERIC_READ value set in the *DesiredAccess* parameter.

If the preceding call to ZwCreateFile set the FILE_NO_INTERMEDIATE_BUFFERING flag in the *CreateOptions* parameter to **NtCreateFile**, the *Length* and *ByteOffset* parameters to **NtReadFile** must be multiples of the sector size.

NtReadFile begins reading from the given *ByteOffset* or the current file position into the given *Buffer*. It terminates the read operation under one of the following conditions:

- The buffer is full because the number of bytes specified by the *Length* parameter has been read. Therefore, no more data can be placed into the buffer without an overflow.
- The end of file is reached during the read operation, so there is no more data in the file to be transferred into the buffer.

If the caller opened the file with the SYNCHRONIZE flag set in *DesiredAccess*, the calling thread can synchronize to the completion of the read operation by waiting on the file handle, *FileHandle*. The handle is signaled each time that an I/O operation that was issued on the handle completes. However, the caller must not wait on a handle that was opened for synchronous file access (FILE_SYNCHRONOUS_IO_NONALERT or FILE_SYNCHRONOUS_IO_ALERT). In this case, **NtReadFile** waits on behalf of the caller and does not return until the read operation is complete. The caller can safely wait on the file handle only if all three of the following conditions are met:

- The handle was opened for asynchronous access (that is, neither FILE_SYNCHRONOUS_IO_**XXX** flag was specified).
- The handle is being used for only one I/O operation at a time.
- **NtReadFile** returned STATUS_PENDING.

A driver should call **NtReadFile** in the context of the system process if any of the following conditions exist:

- The driver created the file handle that it passes to **NtReadFile**.
- **NtReadFile** will notify the driver of I/O completion by means of an event that the driver created.
- **NtReadFile** will notify the driver of I/O completion by means of an APC callback routine that the driver passes to **NtReadFile**.

File and event handles are valid only in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **NtReadFile** in the context of the system process rather than the context of the process that the driver is in.

Likewise, **NtReadFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread that issues the I/O request. If the driver calls **NtReadFile** in the context of a process other than the system one, the APC could be delayed indefinitely, or it might not fire at all.

For more information about working with files, see Using Files in a Driver.

Callers of **NtReadFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "NtReadFile" instead of "**ZwReadFile**".

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, BufAfterReqCompletedIntIoctlA, BufAfterReqCompletedIoctlA, BufAfterReqCompletedReadA, BufAfterReqCompletedWriteA, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

ZwCreateFile

ZwQueryInformationFile

ZwSetInformationFile

ZwWriteFile

NtSetInformationFile function

1/25/2019 • 4 minutes to read • Edit Online

The **NtSetInformationFile** routine changes various kinds of information about a file object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationFile(
  HANDLE                 FileHandle,
  PIO_STATUS_BLOCK       IoStatusBlock,
  PVOID                  FileInformation,
  ULONG                  Length,
  FILE_INFORMATION_CLASS FileInformationClass
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to NtCreateFile or NtOpenFile.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested operation. The **Information** member receives the number of bytes set on the file.

`FileInformation`

Pointer to a buffer that contains the information to set for the file. The particular structure in this buffer is determined by the *FileInformationClass* parameter. Setting any member of the structure to zero tells **NtSetInformationFile** to leave the current information about the file for that member unchanged.

`Length`

The size, in bytes, of the *FileInformation* buffer.

`FileInformationClass`

The type of information, supplied in the buffer pointed to by *FileInformation*, to set for the file. Device and intermediate drivers can specify any of the following FILE_INFORMATION_CLASS values.

***FILEINFORMATIONCLASS* VALUE**	**MEANING**
FileBasicInformation	Change the information that is supplied in a FILE_BASIC_INFORMATION structure. The caller must have opened the file with the FILE_WRITE_ATTRIBUTES flag set in the *DesiredAccess* parameter.
FileCaseSensitiveInformation	Change the information that is supplied in a FILE_CASE_SENSITIVE_INFORMATION structure. The caller must have opened the file with the FILE_WRITE_ATTRIBUTES flag set in the *DesiredAccess* parameter. This value is available starting with Windows 10, version 1803.

FileCaseSensitiveInformationForceAccessCheck	Change the information that is supplied in a FILE_CASE_SENSITIVE_INFORMATION structure. The caller must have opened the file with the FILE_WRITE_ATTRIBUTES flag set in the *DesiredAccess* parameter. This is a special version of the FileCaseSensitiveInformation operation that is used to force the IOManager to perform access checks for the kernel-mode driver, similar to the checks that apply to a user-mode caller. This operation is only recognized by the IOManager and a file system should never receive it. This value is available starting with Windows 10, version 1803.
FileDispositionInformation	Usually, sets the **DeleteFile** member of a FILE_DISPOSITION_INFORMATION to **TRUE**, so the file can be deleted when NtClose is called to release the last open handle to the file object. The caller must have opened the file with the DELETE flag set in the *DesiredAccess* parameter.
FileEndOfFileInformation	Change the current end-of-file information, supplied in a FILE_END_OF_FILE_INFORMATION structure. The operation can either truncate or extend the file. The caller must have opened the file with the FILE_WRITE_DATA flag set in the *DesiredAccess* parameter.
FileIoPriorityHintInformation	Change the current default IRP priority hint for the file handle. The new value is supplied in a FILE_IO_PRIORITY_HINT_INFORMATION structure. **Note** This structure must be 8-byte aligned.
FileLinkInformation	Create a hard link to an existing file, which is specified in a FILE_LINK_INFORMATION structure. Not all file systems support hard links; for example NTFS does while FAT does not.
FilePositionInformation	Change the current file information, which is stored in a FILE_POSITION_INFORMATION structure.
FileRenameInformation	Change the current file name, which is supplied in a FILE_RENAME_INFORMATION structure. The caller must have DELETE access to the file.
FileShortNameInformation	Change the current short file name, which is supplied in a FILE_NAME_INFORMATION structure. The file must be on an NTFS volume, and the caller must have opened the file with the *DesiredAccess* DELETE flag set in the *DesiredAccess* parameter.
FileValidDataLengthInformation	Change the current valid data length for the file, which is supplied in a FILE_VALID_DATA_LENGTH_INFORMATION structure. The file must be on an NTFS volume, and the caller must have opened the file with the FILE_WRITE_DATA flag set in the *DesiredAccess* parameter. Non-administrators and remote users must have the **SeManageVolumePrivilege** privilege.

FileReplaceCompletionInformation	Change or remove the I/O completion port for the specified file handle. The caller supplies a pointer to a FILE_COMPLETION_INFORMATION structure that specifies a port handle and a completion key. If the port handle is non-NULL, this handle specifies a new I/O completion port to associate with the file handle. To remove the I/O completion port associated with the file handle, set the port handle in the structure to NULL. To get a port handle, a user-mode caller can call the CreateIoCompletionPort function.

Return Value

NtSetInformationFile returns STATUS_SUCCESS or an appropriate error status.

Remarks

NtSetInformationFile changes information about a file. It ignores any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by a particular device or file system.

If you set *FileInformationClass* to **FileDispositionInformation**, you can subsequently pass *FileHandle* to NtClose but not to any other **Zw*Xxx*File** routine. Because **FileDispositionInformation** causes the file to be marked for deletion, it is a programming error to attempt any subsequent operation on the handle other than closing it.

If you set *FileInformationClass* to **FilePositionInformation**, and the preceding call to **NtCreateFile** included the FILE_NO_INTERMEDIATE_BUFFERING flag in the *CreateOptions* parameter, certain restrictions on the **CurrentByteOffset** member of the FILE_POSITION_INFORMATION structure are enforced. For more information, see NtCreateFile.

If you set *FileInformationClass* to **FileEndOfFileInformation**, and the **EndOfFile** member of FILE_END_OF_FILE_INFORMATION specifies an offset beyond the current end-of-file mark, **NtSetInformationFile** extends the file and pads the extension with zeros.

For more information about working with files, see Using Files in a Driver.

Callers of **NtSetInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtSetInformationFile**" instead of "**ZwSetInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

CreateIoCompletionPort

FILE_BASIC_INFORMATION

FILE_COMPLETION_INFORMATION

FILE_DISPOSITION_INFORMATION

FILE_END_OF_FILE_INFORMATION

FILE_IO_PRIORITY_HINT_INFORMATION

FILE_LINK_INFORMATION

FILE_NAME_INFORMATION

FILE_POSITION_INFORMATION

FILE_RENAME_INFORMATION

FILE_VALID_DATA_LENGTH_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateFile

ZwOpenFile

ZwQueryInformationFile

NtSetInformationThread function

1/25/2019 • 2 minutes to read • Edit Online

The **NtSetInformationThread** routine sets the priority of a thread.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationThread(
  HANDLE          ThreadHandle,
  THREADINFOCLASS ThreadInformationClass,
  PVOID           ThreadInformation,
  ULONG           ThreadInformationLength
);
```

Parameters

`ThreadHandle`

Handle to the thread object. To create a new thread and get a handle to it, call PsCreateSystemThread. To specify the current thread, use the ZwCurrentThread macro.

`ThreadInformationClass`

One of the system-defined values in the THREADINFOCLASS enumeration (see ntddk.h), **ThreadPriority**, **ThreadBasePriority**, **ThreadPagePriority**, or **ThreadPowerThrottlingState**.

`ThreadInformation`

Pointer to a variable that specifies the information to set.

If **ThreadInformationClass** is **ThreadPriority**, this value must be > LOW_PRIORITY and <= HIGH_PRIORITY.

If **ThreadInformationClass** is **ThreadBasePriority**, this value must fall within the system's valid base-priority range and the original priority class for the given thread. That is, if a thread's priority class is variable, that thread's base priority cannot be reset to a real-time priority value, and vice versa.

If **ThreadInformationClass** is **ThreadPagePriority**, this value is a pointer to a **PAGE_PRIORITY_INFORMATION** structure, see ntddk.h. The **PagePriority** member value must be one of these values.

If **ThreadInformationClass** is **ThreadPowerThrottlingState**, this value is a pointer to a POWER_THROTTLING_THREAD_STATE structure, see ntddk.h. The **PagePriority** member value must be one of these values.

VALUE	MEANING

MEMORY_PRIORITY_VERY_LOW 1	Very low memory priority.
MEMORY_PRIORITY_LOW 2	Low memory priority.
MEMORY_PRIORITY_MEDIUM 3	Medium memory priority.
MEMORY_PRIORITY_BELOW_NORMAL 4	Below normal memory priority.
MEMORY_PRIORITY_NORMAL 5	Normal memory priority. This is the default priority for all threads and processes on the system.

`ThreadInformationLength`

The size, in bytes, of **ThreadInformation**.

Return Value

NtSetInformationThread returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. Possible error codes include STATUS_INFO_LENGTH_MISMATCH or STATUS_INVALID_PARAMETER.

Remarks

NtSetInformationThread can be called by higher-level drivers to set the priority of a thread for which they have a handle.

The caller must have THREAD_SET_INFORMATION access rights for the given thread in order to call this routine.

Usually, device and intermediate drivers that set up driver-created threads call KeSetBasePriorityThread or KeSetPriorityThread from their driver-created threads, rather than calling **NtSetInformationThread**. However, a driver can call **NtSetInformationThread** to raise the priority of a driver-created thread before that thread runs.

Kernel mode drivers can call the **NtSetInformationThread** function with **ThreadPagePriority** to specify a thread's page priority.

To help improve system performance, drivers should use the function with **ThreadPagePriority** to lower the page priority of threads that perform background operations or access files and data that are not expected to be accessed again soon. For example, an anti-malware application might lower the priority of threads involved in scanning files.

To determine the page priority for a thread, call ZwQueryInformationThread.

> **Note** If the call to this function occurs in user mode, you should use the name NtSetInformationThread instead of **ZwSetInformationThread**.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeSetBasePriorityThread

KeSetPriorityThread

PsCreateSystemThread

Using Nt and Zw Versions of the Native System Services Routines

NtSetInformationToken function

1/25/2019 • 3 minutes to read • Edit Online

The **NtSetInformationToken** routine modifies information in a specified token. The calling process must have appropriate access rights to set the information.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationToken(
  HANDLE                  TokenHandle,
  TOKEN_INFORMATION_CLASS TokenInformationClass,
  PVOID                   TokenInformation,
  ULONG                   TokenInformationLength
);
```

Parameters

`TokenHandle`

Handle for an access token in which information is to be modified.

`TokenInformationClass`

A value from the TOKEN_INFORMATION_CLASS enumerated type identifying the type of information to be modified. The possible values for this parameter are listed in the *TokenInformationClass* Value column of the table shown in the description of the *TokenInformation* parameter.

`TokenInformation`

Pointer to a caller-supplied buffer containing the information to be modified in the token. The structure of the information in this buffer depends upon the value of *TokenInformationClass*, as shown in the following table. All structures must be aligned on a 32-bit boundary.

***TOKENINFORMATIONCLASS* VALUE**	**EFFECT ON *TOKENINFORMATION* BUFFER**
TokenDefaultDacl	The buffer contains a TOKEN_DEFAULT_DACL structure specifying the default DACL for newly created objects. TOKEN_ADJUST_DEFAULT access is required to set this information. The buffer contents are not validated for structural correctness or consistency.
TokenGroups	Not a valid information class. This information is read-only.
TokenOwner	The buffer contains a TOKEN_OWNER structure specifying the default owner SID for newly created objects. TOKEN_ADJUST_DEFAULT access is required to set this information. The owner values that may be specified are restricted to the user and group IDs with an attribute indicating they can be assigned as the owner of objects.

TokenPrimaryGroup	The buffer contains a TOKEN_PRIMARY_GROUP structure specifying the default primary group **SID** for newly created objects. TOKEN_ADJUST_DEFAULT access is required to set this information. Must be one of the group IDs already in the token.
TokenPrivileges	Not a valid information class. This information is read-only.
TokenSource	Not a valid information class. This information is read-only.
TokenStatistics	Not a valid information class. This information is read-only.
TokenUser	Not a valid information class. This information is read-only.

`TokenInformationLength`

Size, in bytes, of the structure passed in the *TokenInformation* buffer. Must be greater than or equal to the minimum value given in the following table.

TOKENINFORMATIONCLASS VALUE	MINIMUM *TOKENINFORMATIONLENGTH*
TokenDefaultDacl	**sizeof**(**TOKEN_DEFAULT_DACL**)
TokenOwner	**sizeof**(**TOKEN_OWNER**)
TokenPrimaryGroup	**sizeof**(**TOKEN_PRIMARY_GROUP**)

Return Value

NtSetInformationToken returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_ACCESS_DENIED	*TokenHandle* did not have the required access.
STATUS_ALLOTTED_SPACE_EXCEEDED	The space allotted for storage of the default discretionary access control and the primary group ID is not large enough to accept the new value of one of these fields.

STATUS_INFO_LENGTH_MISMATCH	The value of *TokenInformationLength* was less than the required minimum.
STATUS_INSUFFICIENT_RESOURCES	The specified default owner's security information could not be captured.
STATUS_INVALID_HANDLE	*TokenHandle* was not a valid handle.
STATUS_INVALID_INFO_CLASS	*TokenInformationClass* was not a valid token information class.
STATUS_INVALID_OWNER	The caller cannot set the specified ID to be an owner (or default owner) of an object.

STA TUS _IN VAL ID_ PRI MA RY_ GR OU P	The caller cannot set the specified ID to be the primary group of an object.
STA TUS _IN VAL ID_S ID	The specified default owner's security information was not valid.
STA TUS _OB JEC T_T YPE _MI SM ATC H	*TokenHandle* was not a token handle.

Remarks

For more information about security and access control, see the documentation on these topics in the Windows SDK.

> **Note** If the call to the **NtSetInformationToken** function occurs in user mode, you should use the name "NtSetInformationToken" instead of "**ZwSetInformationToken**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PsDereferenceImpersonationToken

PsDereferencePrimaryToken

SECURITY_IMPERSONATION_LEVEL

SID

SeQueryAuthenticationIdToken

SeQuerySubjectContextToken

SeTokenIsAdmin

SeTokenIsRestricted

TOKEN_DEFAULT_DACL

TOKEN_GROUPS

TOKEN_INFORMATION_CLASS

TOKEN_OWNER

TOKEN_PRIMARY_GROUP

TOKEN_PRIVILEGES

TOKEN_SOURCE

TOKEN_STATISTICS

TOKEN_TYPE

TOKEN_USER

Using Nt and Zw Versions of the Native System Services Routines

NtQueryInformationToken

NtSetQuotaInformationFile function

1/25/2019 • 2 minutes to read • Edit Online

The **NtSetQuotaInformationFile** routine changes quota entries for the volume associated with the *FileHandle* parameter. All of the quota entries in the specified buffer are applied to the volume.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetQuotaInformationFile(
  HANDLE           FileHandle,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length
);
```

Parameters

`FileHandle`

A handle for the file object that represents the file or volume for which the quota information is to be modified.

`IoStatusBlock`

The address of the caller's I/O status block.

`Buffer`

A buffer containing the new quota entries that should be applied to the volume. The quota information must be formatted as one or more FILE_QUOTA_INFORMATION structures. The **NextEntryOffset** field in the **FILE_QUOTA_INFORMATION** structure contains the offset, in bytes, of the next quota entry in the list. If there are no more entries after the current one, this member is zero.

`Length`

The length in bytes of the buffer.

Return Value

The **NtSetQuotaInformationFile** routine returns STATUS_SUCCESS or an appropriate NTSTATUS value such as one of the following:

RETURN CODE	DESCRIPTION

STATUS_INSUFFICIENT_RESOURCES	There were insufficient resources to complete the operation. This is an error code.
STATUS_INVALID_DEVICE_REQUEST	Quotas are not enabled on the volume. This is an error code.
STATUS_MEDIA_WRITE_PROTECTED	The volume is read only. This is an error code.

Remarks

The **NtSetQuotaInformationFile** routine applies all of the quota entries in the specified *Buffer* parameter to the volume.

The **IoCheckQuotaBufferValidity** function can check whether the specified quota buffer passed as the *Buffer* parameter is valid.

A call to **NtSetQuotaInformationFile** will result in an IRP_MJ_SET_QUOTA request being sent to the device object that is associated with the file object whose handle is stored in the *FileHandle* parameter.

If the underlying file system does not support quota information (FAT and CDFS file systems, for example), **NtSetQuotaInformationFile** will fail returning STATUS_INVALID_DEVICE_REQUEST.

> **Note** If the call to the **NtSetQuotaInformationFile** function occurs in user mode, you should use the name "NtSetQuotaInformationFile" instead of "**ZwSetQuotaInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows operating systems.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_GET_QUOTA_INFORMATION

FILE_QUOTA_INFORMATION

IO_STATUS_BLOCK

IRP_MJ_QUERY_QUOTA

IRP_MJ_SET_QUOTA

IoCheckQuotaBufferValidity

Using Nt and Zw Versions of the Native System Services Routines

NtQueryQuotaInformationFile

NtSetSecurityObject function

1/25/2019 • 2 minutes to read • Edit Online

The **NtSetSecurityObject** routine sets an object's security state.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetSecurityObject(
  HANDLE               Handle,
  SECURITY_INFORMATION SecurityInformation,
  PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Parameters

`Handle`

Handle for the object whose security state is to be set. This handle must have the access specified in the Meaning column of the table shown in the description of the *SecurityInformation* parameter.

`SecurityInformation`

SECURITY_INFORMATION value specifying the information to be set. Can be a combination of one or more of the following.

VALUE	MEANING
DACL_SECURITY_INFORMATION	Indicates the discretionary access control list (DACL) of the object is to be set. Requires WRITE_DAC access.
GROUP_SECURITY_INFORMATION	Indicates the primary group identifier of the object is to be set. Requires WRITE_OWNER access.
OWNER_SECURITY_INFORMATION	Indicates the owner identifier of the object is to be set. Requires WRITE_OWNER access.
SACL_SECURITY_INFORMATION	Indicates the system ACL (SACL) of the object is to be set. Requires ACCESS_SYSTEM_SECURITY access.

`SecurityDescriptor`

Pointer to the security descriptor to be set for the object.

Return Value

NtSetSecurityObject returns STATUS_SUCCESS or an appropriate error status. Possible error status codes include the following:

RETURN CODE	DESCRIPTION

STATUS_ACCESS_DENIED	*Handle* does not have the required access rights.
STATUS_ACCESS_VIOLATION	*SecurityDescriptor* is a **NULL** pointer.
STATUS_INSUFFICIENT_RESOURCES	The object's security descriptor could not be captured.
STATUS_INVALID_ACL	The object's security descriptor contains an invalid ACL.
STATUS_INVALID_HANDLE	*Handle* is not a valid handle.

STA TUS _IN VAL ID_S ECU RIT Y_D ESC R	*SecurityDescriptor* does not point to a valid security descriptor.
STA TUS _IN VAL ID_S ID	The object's security descriptor contains an invalid **SID**.
STA TUS _OB JEC T_T YPE _MI SM ATC H	*Handle* is not a handle of the expected type.
STA TUS _UN KN OW N_R EVIS ION	The revision level of the object's security descriptor is unknown or is not supported.

Remarks

A security descriptor can be in absolute or self-relative form. In self-relative form, all members of the structure are located contiguously in memory. In absolute form, the structure only contains pointers to the members. For more information, see "Absolute and Self-Relative Security Descriptors" in the Security section of the Windows SDK documentation.

For more information about security and access control, see the documentation on these topics in the Windows SDK.

Minifilters should use FltSetSecurityObject instead of **ZwSetSecurityObject**.

Callers of **NtSetSecurityObject** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

Note If the call to the **NtSetSecurityObject** function occurs in user mode, you should use the name "NtSetSecurityObject" instead of "**ZwSetSecurityObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltSetSecurityObject

SECURITY_DESCRIPTOR

SECURITY_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

NtQuerySecurityObject

NtUnlockFile function

1/25/2019 • 2 minutes to read • Edit Online

The **NtUnlockFile** routine unlocks a byte-range lock in a file.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtUnlockFile(
  HANDLE           FileHandle,
  PIO_STATUS_BLOCK IoStatusBlock,
  PLARGE_INTEGER   ByteOffset,
  PLARGE_INTEGER   Length,
  ULONG            Key
);
```

Parameters

`FileHandle`

A handle for the file object that represents the file whose byte range is to be unlocked.

`IoStatusBlock`

A pointer to an IO_STATUS_BLOCK structure that contains the final status.

`ByteOffset`

A pointer to a variable that specifies the starting byte offset for the byte range to be unlocked.

`Length`

A pointer to a variable that specifies the length, in bytes, of the byte range to unlock.

`Key`

The caller-assigned value used to describe groups of related locks. This value should be set to zero.

Return Value

The **NtUnlockFile** routine returns STATUS_SUCCESS on success or an appropriate NTSTATUS value. Possible NTSTATUS values include:

RETURN CODE	DESCRIPTION
STATUS_RANGE_NOT_LOCKED	The byte range specified is not locked.

Remarks

The **NtUnlockFile** routine takes a range of bytes as specified by the *ByteOffset* and *Length* arguments. This range must be identical to a range of bytes in the file that was previously locked with a single call to the **NtUnlockFile** routine. It is not possible to unlock two previously locked adjacent ranges with a single call to **NtUnlockFile**. It is also not possible to unlock part of a range that was previously locked with a single call to the **NtUnlockFile** routine.

Callers of **NtUnlockFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to the **NtUnlockFile** function occurs in user mode, you should use the name "NtUnlockFile" instead of "**ZwUnlockFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, FltKernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

NtLockFile

NtWriteFile function

1/25/2019 • 5 minutes to read • Edit Online

The **ZwWriteFile** routine writes data to an open file.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtWriteFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to NtCreateFile or NtOpenFile.

`Event`

Optionally, a handle to an event object to set to the signaled state after the write operation completes. Device and intermediate drivers should set this parameter to **NULL**.

`ApcRoutine`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`ApcContext`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested write operation. The **Information** member receives the number of bytes actually written to the file.

`Buffer`

Pointer to a caller-allocated buffer that contains the data to write to the file.

`Length`

The size, in bytes, of the buffer pointed to by *Buffer*.

`ByteOffset`

Pointer to a variable that specifies the starting byte offset in the file for beginning the write operation. If *Length* and *ByteOffset* specify a write operation past the current end-of-file mark, **NtWriteFile** automatically extends the file and updates the end-of-file mark; any bytes that are not explicitly written between such old and new end-of-file

marks are defined to be zero.

If the call to NtCreateFile set only the *DesiredAccess* flag FILE_APPEND_DATA, *ByteOffset* is ignored. Data in the given *Buffer*, for *Length* bytes, is written starting at the current end of file.

If the call to NtCreateFile set either of the *CreateOptions* flags, FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **NtWriteFile** can specify that the current file position offset be used instead of an explicit *ByteOffset* value. This specification can be made by using one of the following methods:

- Specify a pointer to a LARGE_INTEGER value with the **HighPart** member set to -1 and the **LowPart** member set to the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a **NULL** pointer for *ByteOffset*.

NtWriteFile updates the current file position by adding the number of bytes written when it completes the write operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **NtWriteFile**. Doing this automatically changes the current file position to that *ByteOffset*value, performs the write operation, and then updates the position according to the number of bytes actually written. This technique gives the caller atomic seek-and-write service.

It is also possible to cause a write operation to start at the current end of file by specifying for *ByteOffset* a pointer to a LARGE_INTEGER value with **HighPart** set to -1 and **LowPart** set to FILE_WRITE_TO_END_OF_FILE. This works regardless of whether the I/O Manager is maintaining the current file position.

`Key`

Device and intermediate drivers should set this pointer to **NULL**.

Return Value

NtWriteFile returns STATUS_SUCCESS on success or the appropriate NTSTATUS error code on failure.

Remarks

Callers of **NtWriteFile** must have already called NtCreateFile with the FILE_WRITE_DATA, FILE_APPEND_DATA, or GENERIC_WRITE flag set in the *DesiredAccess* parameter. Note that having only FILE_APPEND_DATA access to a file does not allow the caller to write anywhere in the file except at the current end-of-file mark, while having FILE_WRITE_DATA access to a file does not preclude the caller from writing to or beyond the end of a file.

If the preceding call to NtCreateFile set the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING, the *Length* and *ByteOffset* parameters to **NtWriteFile** must be an integral of the sector size. For more information, see **NtCreateFile**.

NtWriteFile begins the write operation to the file at *ByteOffset*, at the current file position, or at the end-of-file mark. It terminates the write operation when it has written *Length* bytes from *Buffer*. If necessary, it extends the length of the file and resets the end-of-file mark.

If the caller opened the file with the *DesiredAccess* SYNCHRONIZE flag set, the caller can wait for this routine to set the given *FileHandle* to the signaled state.

Drivers should call **NtWriteFile** in the context of the system process in three cases:

1. The driver creates the file handle that it passes to **NtWriteFile**.
2. **NtWriteFile** notifies the driver of I/O completion by means of an event created by the driver.

3. **NtWriteFile** notifies the driver of I/O completion by means of an APC callback routine that the driver passes to **NtWriteFile**.

File and event handles are only valid in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **NtWriteFile** in the context of the system process instead of the process context that the driver is in.

Likewise, **NtWriteFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread issuing the I/O request. If the driver calls **NtWriteFile** in the context of a process other than the system process, the APC could be delayed indefinitely, or it might not fire at all as the originating thread may never enter an alertable wait state.

For more information about working with files, see Using Files in a Driver.

Callers of **NtWriteFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "NtWriteFile" instead of "**ZwWriteFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

ZwCreateFile

ZwQueryInformationFile

ZwReadFile

ZwSetInformationFile

PoCallDriver function

1/25/2019 • 2 minutes to read • Edit Online

The **PoCallDriver** routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.)

Syntax

```
NTSTATUS PoCallDriver(
  PDEVICE_OBJECT        DeviceObject,
  __drv_aliasesMem PIRP Irp
);
```

Parameters

`DeviceObject`

A pointer to the driver-created DEVICE_OBJECT to which the IRP is to be routed.

`Irp`

A pointer to an IRP.

Return Value

PoCallDriver returns STATUS_SUCCESS to indicate success. It returns STATUS_PENDING if it has queued the IRP.

Remarks

Beginning with Windows Vista, drivers should call IoCallDriver, not **PoCallDriver** to pass a power IRP to the next-lower driver. However, on Windows Server 2003, Windows XP, and Windows 2000, drivers must call **PoCallDriver**, not **IoCallDriver** to pass a power IRP to the next-lower driver. On Windows Server 2003, Windows XP, an Windows 2000, drivers must also call PoStartNextPowerIrp before calling **PoCallDriver**.

A driver that requires a new IRP should call PoRequestPowerIrp. A driver must not allocate its own power IRP.

When passing a power IRP down to the next-lower driver, the caller should use IoSkipCurrentIrpStackLocation or IoCopyCurrentIrpStackLocationToNext to set the IRP stack location, then call **PoCallDriver**. Use **IoCopyCurrentIrpStackLocationToNext** if processing the IRP requires setting an IoCompletion routine, or **IoSkipCurrentStackLocation** if no *IoCompletion* routine is required.

When a device is powering up, its drivers must set *IoCompletion* routines to perform start-up tasks (initializing the device, restoring context, and so on) after the bus driver has set the device in the working state. Set *IoCompletion* routines before calling **PoCallDriver**.

When a device is powering down, its drivers must perform necessary power-down tasks before passing the IRP to the next lower driver. After the IRP has reached the bus driver, the device will be powered off and its drivers will no longer have access to it. On Windows Server 2003, Windows XP, and Windows 2000, an *IoCompletion* routine that is associated with a power-down IRP is only required to call **PoStartNextPowerIrp**.

Only one inrush IRP can be active in the system at a time. When passing a power-up IRP for a device that requires

inrush current (in other words, the DO_POWER_INRUSH flag is set in the device object), **PoCallDriver** checks whether another inrush IRP is already active. If so, **PoCallDriver** queues the current IRP for handling after the previous IRP completes, and then returns STATUS_PENDING. For more information about inrush IRPs, see Setting Device Object Flags for Power Management.

If an IRP_MN_SET_POWER or IRP_MN_QUERY_POWER request is already active for *DeviceObject*, **PoCallDriver** queues this IRP and returns STATUS_PENDING.

On Windows 2000 and later systems, pageable drivers (the DO_POWER_PAGABLE flag is set in the device object) must call **PoCallDriver** at IRQL = PASSIVE_LEVEL. Drivers that cannot be paged (DO_POWER_PAGABLE is not set in the device object) or that require inrush current (DO_POWER_INRUSH is set in the device object) can call **PoCallDriver** at IRQL = PASSIVE_LEVEL or DISPATCH_LEVEL.

On Windows 98/Me, all drivers call **PoCallDriver** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	CompleteRequestStatusCheck, CompletionRoutineRegistered, DeleteDevice, ForwardedAtBadIrql, ForwardedAtBadIrqlAllocate, ForwardedAtBadIrqlFsdAsync, ForwardedAtBadIrqlFsdSync, IoAllocateForward, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildFsdForward, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, IoSetCompletionRoutineExCheck, IrpProcessingComplete, LowerDriverReturn, MarkDevicePower, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, PnpIrpCompletion, PowerDownFail, PowerUpFail, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove2, RemoveLockMnSurpriseRemove, RemoveLockQueryMnRemove, TargetRelationNeedsRef, WmiForward, HwStorPortProhibitedDDIs

See Also

IRP

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IoCallDriver

IoCompletion

IoCopyCurrentIrpStackLocationToNext

IoSkipCurrentIrpStackLocation

PoRequestPowerIrp

PoStartNextPowerIrp

PoClearPowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoClearPowerRequest** routine decrements the count for the specified power request type.

Syntax

```
NTSTATUS PoClearPowerRequest(
  PVOID              PowerRequest,
  POWER_REQUEST_TYPE Type
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

`Type`

The type of the power request. Set this parameter to the following POWER_REQUEST_TYPE enumeration value:

- **PowerRequestSystemRequired**

Return Value

PoClearPowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_UNSUPPORTED	Parameter *Type* is set to an unsupported value.

Remarks

A driver can call the PoSetPowerRequest routine to request that the power manager override several types of default power behavior, which are specified as POWER_REQUEST_TYPE enumeration values. To restore the default behavior, the driver cancels the request by calling the **PoClearPowerRequest** routine.

The power manager maintains a count of the active requests for each power request type. The **PoSetPowerRequest** routine increments the count for the specified power request type by one. The **PoClearPowerRequest** routine decrements the count by one. A nonzero count indicates that requests from one or more components are active. After the count decrements to zero, the power manager restores the default behavior for the specified power request type.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

POWER_REQUEST_TYPE

PoCreatePowerRequest

PoSetPowerRequest

PoCreatePowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoCreatePowerRequest** routine creates a power request object.

Syntax

```
NTSTATUS PoCreatePowerRequest(
  PVOID                    *PowerRequest,
  PDEVICE_OBJECT           DeviceObject,
  PCOUNTED_REASON_CONTEXT Context
);
```

Parameters

`PowerRequest`

A pointer to a location into which the routine writes a pointer to the newly created power request object. If the call fails, the routine writes **NULL** to this location.

`DeviceObject`

A pointer to the device object of the caller (a DEVICE_OBJECT structure).

`Context`

A pointer to a COUNTED_REASON_CONTEXT structure that describes why the caller is creating the power request object. This parameter is optional and can be set to **NULL**.

Return Value

PoCreatePowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *DeviceObject* parameter is **NULL**.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	There is not enough memory available to create a power request object.

Remarks

This routine creates a power request object. To enable power requests, the caller should create one power request object and use that object for all calls to the PoSetPowerRequest and PoClearPowerRequest routines.

A driver can use power requests to override certain aspects of the computer's default power behavior. For example, a driver for a TV receiver device can use power requests to prevent the power manager from automatically blanking the display during extended periods of time in which no user interaction occurs.

When the power request object is no longer needed, the caller must delete the object by calling the PoDeletePowerRequest routine. The driver must delete the power request object before it deletes the device object that was used to create the power request object.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

PoClearPowerRequest

PoDeletePowerRequest

PoSetPowerRequest

PoDeletePowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoDeletePowerRequest** routine deletes a power request object.

Syntax

```
void PoDeletePowerRequest(
  PVOID PowerRequest
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

Return Value

None

Remarks

The driver must delete the power request object before it deletes the device object that was used to create the power request object.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoCreatePowerRequest

PoEndDeviceBusy function

1/25/2019 • 2 minutes to read • Edit Online

The **PoEndDeviceBusy** routine marks the end of a period of time in which the device is busy.

Syntax

```
void PoEndDeviceBusy(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoEndDeviceBusy**.

Return Value

None

Remarks

The PoStartDeviceBusy and **PoEndDeviceBusy** routines mark the start and end of a time period in which a device is busy. Each call to **PoStartDeviceBusy** must be followed by a corresponding call to **PoEndDeviceBusy**. For more information about how these routines work together, see PoStartDeviceBusy.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

PoRegisterDeviceForIdleDetection

PoStartDeviceBusy

PoQueryWatchdogTime function

1/25/2019 • 2 minutes to read • Edit Online

The **PoQueryWatchdogTime** routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.

Syntax

```
BOOLEAN PoQueryWatchdogTime(
  PDEVICE_OBJECT Pdo,
  PULONG         SecondsRemaining
);
```

Parameters

`Pdo`

A pointer to a physical device object (PDO). This parameter points to a DEVICE_OBJECT structure that represents a physical device.

`SecondsRemaining`

A pointer to a location into which the routine writes the time, in seconds, that remains before the next power watchdog time-out is set to occur.

Return Value

PoQueryWatchdogTime returns **TRUE** if a watchdog-enabled power IRP is currently assigned to the device stack. Otherwise, it returns **FALSE**.

Remarks

This routine enables kernel-mode drivers to monitor watchdog time-out counters that the power manager has enabled to keep track of power IRPs that it has issued. If one or more watchdog time-out counters are currently enabled, the routine returns **TRUE** and provides the amount of time that remains before the next time-out.

For example, a driver that experiences delays when shutting down a device can call this routine to determine how much time remains before the driver must respond to a power IRP to prevent a controlled shutdown (a bug check) of the operating system.

The power manager sets a watchdog time-out counter when it issues a power IRP to the device stack. The time-out period for this counter is typically several minutes. If a device in the stack is unresponsive and causes the IRP to stall for the time-out period, the power manager treats this condition as an unrecoverable error and initiates a controlled shutdown of the operating system.

If more than one power watchdog time-out is currently enabled, the routine sets **SecondsRemaining* to the time that remains to the next time-out.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DEVICE_OBJECT

PoRegisterDeviceForIdleDetection function

1/25/2019 • 3 minutes to read • Edit Online

The **PoRegisterDeviceForIdleDetection** routine enables or cancels idle detection and sets idle time-out values for a device.

Syntax

```
PULONG PoRegisterDeviceForIdleDetection(
  PDEVICE_OBJECT     DeviceObject,
  ULONG              ConservationIdleTime,
  ULONG              PerformanceIdleTime,
  DEVICE_POWER_STATE State
);
```

Parameters

`DeviceObject`

A pointer to the driver-created DEVICE_OBJECT for the device. On Windows 2000 and later systems, this parameter can point to a physical device object (PDO) or a functional device object (FDO). On Windows 98/Me, this parameter must point to the PDO of the underlying device.

`ConservationIdleTime`

Sets the time-out value (in seconds) to apply when the system power policy optimizes for energy conservation. Specify zero to disable idle detection when conservation policy is in effect.

`PerformanceIdleTime`

Sets the time-out value (in seconds) to apply when the system power policy optimizes for performance. Specify zero to disable idle detection when performance policy is in effect.

`State`

Specifies the device power state to be requested in an IRP_MN_SET_POWER request when either *ConservationIdleTime* or *PerformanceIdleTime* has been met. Possible values are the DEVICE_POWER_STATE values.

Return Value

PoRegisterDeviceForIdleDetection returns a pointer to the idle counter to indicate that idle detection has been enabled. It returns **NULL** to indicate that idle detection has been disabled, that an idle counter could not be allocated, or that one or both of the time-out values were invalid.

Remarks

PoRegisterDeviceForIdleDetection enables drivers to use the idle detection mechanism provided by the power manager. Drivers call **PoRegisterDeviceForIdleDetection** for any of the following reasons:

- To enable idle detection for the device and set initial idle time-out values
- To change the idle time-out values for a device

- To disable idle detection for a device

After enabling a device for idle detection, a driver calls PoSetDeviceBusy whenever the device is in use, passing the non-**NULL** idle pointer returned by **PoRegisterDeviceForIdleDetection**. Calling **PoSetDeviceBusy** restarts the idle countdown. Note that a driver must not pass a **NULL** pointer to **PoSetDeviceBusy**.

Whenever the device satisfies the current idle time-out value, the power manager sends an **IRP_MN_SET_POWER** request to the top of the device stack, specifying device power state *State*. In response to the IRP, each driver performs any device-specific tasks required before the power state transition, then passes the IRP to the next-lower driver. When the IRP reaches the bus driver, that driver puts the device in the requested lower power state and completes the IRP.

PoRegisterDeviceForIdleDetection sets time-out values for both conservation and performance. The *ConservationIdleTime* value applies when the system power policy optimizes for conservation; the *PerformanceIdleTime* value applies when the system power policy optimizes for performance. Typically, the applicable policy depends upon the power source: when running with AC power, the system optimizes for performance, and when running off a battery, the system optimizes for conservation.

Certain devices can specify time-out values of -1 to use the standard power policy time-outs for their device class. The standard time-out values provide for better system integration for supported standard device classes. At present, WDM supports this feature for devices of type FILE_DEVICE_DISK and FILE_DEVICE_MASS_STORAGE. **PoRegisterDeviceForIdleDetection** returns **NULL** if -1 is specified for a device of an unsupported type. (For information about device types, see Specifying Device Types.)

Only one idle detection can be set per device. Subsequent calls to **PoRegisterDeviceForIdleDetection** change the idle detection values.

If both *ConservationIdleTime* and *PerformanceIdleTime* are zero, this routine cancels all idle detection for the device and returns **NULL**.

PoRegisterDeviceForIdleDetection can free a driver from the need to perform its own idle detection. However, drivers can also implement their own idle detection.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP

IRP_MN_SET_POWER

PoSetDeviceBusy

PoRegisterPowerSettingCallback function

1/25/2019 • 4 minutes to read • Edit Online

The **PoRegisterPowerSettingCallback** routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.

Syntax

```
NTSTATUS PoRegisterPowerSettingCallback(
  PDEVICE_OBJECT          DeviceObject,
  LPCGUID                 SettingGuid,
  PPOWER_SETTING_CALLBACK Callback,
  PVOID                   Context,
  PVOID                   *Handle
);
```

Parameters

`DeviceObject`

A pointer to a DEVICE_OBJECT structure that is associated with the caller of this routine. This parameter is optional. It is used internally only for debugging purposes. If this parameter is not supplied, it must be set to **NULL**.

`SettingGuid`

A pointer to the GUID that represents the power setting for this registration. When the specified power setting changes, the power manager calls the callback routine to notify the driver of the change and to supply the new value of the setting. For more information, see Remarks.

`Callback`

A pointer to a caller-implemented power-setting callback routine that the power manager calls when the specified power setting changes. For the functional prototype for the callback routine, see Power-Setting Callback, below.

`Context`

A pointer to the context for the callback routine. This parameter is optional. It is provided so that a driver or device context can be passed to the callback routine. If this parameter is not used, it must be set to **NULL**.

`Handle`

A handle that the power manager uses to represent the callback routine. A driver must subsequently supply this handle in a call to PoUnregisterPowerSettingCallback to unregister the callback routine.

Return Value

PoRegisterPowerSettingCallback returns one of the following:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The routine registered the callback routine.
STATUS_INSUFFICIENT_RESOURCES	The routine could not allocate the system resources that are required to register the callback routine.

Remarks

A driver calls **PoRegisterPowerSettingCallback** to register a callback routine with the power manager. The power manager subsequently calls this callback routine to notify the driver after there is a change to the specified power setting. In addition, the power manager initializes the power setting of the driver by immediately calling the callback routine and passing the current value of the power setting. The power manager initializes the power setting of the driver this way regardless of whether the power setting has actually changed.

A driver should call **PoRegisterPowerSettingCallback** for each power setting that the driver needs to monitor. Drivers should call this routine in their DriverEntry routine during initialization. Typically, most drivers pass a pointer to a device extension in the *Context* parameter.

To unregister a power-setting callback, call the PoUnregisterPowerSettingCallback routine.

Typically, Kernel-Mode Driver Framework (KMDF) drivers should call **PoRegisterPowerSettingCallback** from their EvtDeviceSelfManagedIoInit callback function, and should call **PoUnregisterPowerSettingCallback** from their EvtDeviceSelfManagedIoCleanup callback function. These drivers should not call **PoRegisterPowerSettingCallback** from their EvtDriverDeviceAdd callback function; otherwise, the power-setting callback routine might be called before the driver stack is completely built.

The callback routine that is registered for a particular power setting is called when a transition in power state occurs that changes the value of the setting, or when the power manager changes the value of the setting. For example, if *SettingGuid* points to the GUID value GUID_LIDSWITCH_STATE_CHANGE, the callback routine is called when the lid to a laptop computer clicks open or closed. The *Value* parameter passed to the callback routine in this example points to a ULONG value that is 1 if the state of the lid switch changed from closed to open, and is 0 if the state of the lid switch changed from open to closed. For more information, see the power-setting GUID definitions and extensive comments in the Wdm.h header file.

The initial call to a callback routine might occur immediately, before the **PoRegisterPowerSettingCallback** call that registers the routine returns.

PoRegisterPowerSettingCallback can be called only at IRQL = PASSIVE_LEVEL.

Power-Setting Callback

The function prototype of the power-setting callback routine is as follows:

```
NTSTATUS
POWER_SETTING_CALLBACK (
  _In_ LPCGUID SettingGuid,
  _In_ PVOID Value,
  _In_ ULONG ValueLength,
  _Inout_opt_ PVOID Context
);
```

The power-setting callback parameters are:

SettingGuid
A pointer to a GUID that represents the power setting that changed. Power settings and their corresponding GUIDs are defined in Wdm.h.

Value A pointer to the new value of the power setting that changed.

ValueLength A value of type ULONG that specifies the size, in bytes, of the new power setting value.

Context
The pointer to the context that a driver supplied in the call to **PoRegisterPowerSettingCallback** that registered the callback routine.

The power manager calls a power-setting callback at IRQL = PASSIVE_LEVEL. #### Examples To define a power-setting callback routine, you must first provide a function declaration that identifies the type of callback function you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system. For example, to define a power-setting callback routine that is named `MyPowerSettingCallback`, use the POWER_SETTING_CALLBACK type as shown in this code example:

```
POWER_SETTING_CALLBACK MyPowerSettingCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyPowerSettingCallback(
    LPCGUID SettingGuid,
    PVOID Value,
    ULONG ValueLength,
    PVOID Context
    )
  {
      // Function body
  }
```

The POWER_SETTING_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the POWER_SETTING_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For

information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)

See Also

DriverEntry

EvtDeviceSelfManagedIoCleanup

EvtDeviceSelfManagedIoInit

EvtDriverDeviceAdd

PoUnregisterPowerSettingCallback

PoRegisterSystemState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoRegisterSystemState** routine registers the system as busy due to certain activity.

Syntax

```
PVOID PoRegisterSystemState(
  PVOID           StateHandle,
  EXECUTION_STATE Flags
);
```

Parameters

`StateHandle`

A pointer to a caller-supplied buffer for a registration state handle. The size, in bytes, of the buffer is **sizeof**(ULONG). If **NULL**, this is a new registration. If non-**NULL**, this parameter points to a handle that was returned by a previous call to **PoRegisterSystemState**.

`Flags`

Indicates the type of activity, as specified by a bitwise OR of one or more of the following values:

ES_SYSTEM_REQUIRED

The system is not idle, regardless of apparent load.

ES_DISPLAY_REQUIRED

Use of the display is required.

ES_USER_PRESENT

A user is present.

ES_CONTINUOUS

The settings are continuous and should remain in effect until explicitly changed.

Return Value

PoRegisterSystemState returns a handle to be used later to change or unregister the system busy state. It returns **NULL** if the handle could not be allocated.

Remarks

PoRegisterSystemState registers the system busy state as indicated by the flags. The registration persists until the caller explicitly changes it with another call to **PoRegisterSystemState** or cancels it with a call to PoUnregisterSystemState.

The *Flags* parameter specifies the type of activity in progress. Drivers can specify any combination of the flags.

Setting ES_CONTINUOUS makes the busy state persist until a driver explicitly changes or cancels it by calling **PoRegisterSystemState** or **PoUnregisterSystemState**.

A driver can set the system busy state to request that the power manager avoid system power state transitions out

of the system working state (S0) while driver activity is occurring. Note, however, that under some circumstances (such as a critically low battery) the power manager may override this request and put the system to sleep anyway.

To set the system power state, call PoSetSystemState.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoSetSystemState

PoUnregisterSystemState

PoSetDeviceBusyEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetDeviceBusyEx** routine notifies the power manager that the device associated with the specified idle counter is busy.

Syntax

```
void PoSetDeviceBusyEx(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoSetDeviceBusyEx**.

Return Value

None

Remarks

This routine is a direct replacement for the PoSetDeviceBusy macro. If you are writing new driver code for Windows Vista with Service Pack 1 (SP1) and later versions of Windows, call **PoSetDeviceBusyEx** instead of **PoSetDeviceBusy**.

A driver calls the **PoSetDeviceBusyEx** and **PoRegisterDeviceForIdleDetection** routines to enable system idle detection for its device. If a device that is registered for idle detection stays idle for the driver-specified time-out period, the power manager sends an IRP_MN_SET_POWER request to put the device in a requested sleep state.

PoSetDeviceBusyEx reports that the device is busy, so that the power manager can restart its idle countdown. If the device is in a sleep state, **PoSetDeviceBusyEx** does not change the state of the device. That is, it does not cause the system to send an **IRP_MN_SET_POWER** request to awaken the device.

PoSetDeviceBusyEx is designed for use with I/O operations that are relatively brief compared to the time-out period of the idle counter. For longer operations that might exceed this period, use the PoStartDeviceBusy and PoEndDeviceBusy routines instead.

A driver that makes multiple requests for brief I/O operations should call **PoSetDeviceBusyEx** for every I/O request that it makes.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1.

Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IRP_MN_SET_POWER

PoEndDeviceBusy

PoRegisterDeviceForIdleDetection

PoSetDeviceBusy

PoStartDeviceBusy

PoSetPowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetPowerRequest** routine increments the count for the specified power request type.

Syntax

```
NTSTATUS PoSetPowerRequest(
  PVOID             PowerRequest,
  POWER_REQUEST_TYPE Type
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

`Type`

The type of power request. Set this parameter to the following POWER_REQUEST_TYPE enumeration value:

- **PowerRequestSystemRequired**

Return Value

PoSetPowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_UNSUPPORTED_TYPE	The *Type* parameter is set to an unsupported value.

Remarks

A driver can call the **PoSetPowerRequest** routine to request that the power manager override several types of default power behavior, which are specified as POWER_REQUEST_TYPE enumeration values. To restore the default behavior, the driver cancels the request by calling the PoClearPowerRequest routine.

The power manager maintains a count of the active requests for each power request type. The **PoSetPowerRequest** routine increments the count for the specified power request type by one. The **PoClearPowerRequest** routine decrements the count by one. A nonzero count indicates that requests from one or more components are active. After the count decrements to zero, the computer reverts to the default behavior for the specified power request type.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

POWER_REQUEST_TYPE

PoClearPowerRequest

PoCreatePowerRequest

PoSetPowerState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetPowerState** routine notifies the system of a change in the device power state for a device.

Syntax

```
POWER_STATE PoSetPowerState(
  PDEVICE_OBJECT   DeviceObject,
  POWER_STATE_TYPE Type,
  POWER_STATE      State
);
```

Parameters

`DeviceObject`

A pointer to the target DEVICE_OBJECT.

`Type`

Indicates a POWER_STATE_TYPE value. Drivers must specify **DevicePowerState**.

`State`

Specifies the power state to be set. Drivers must specify a DEVICE_POWER_STATE value.

Return Value

On Windows 2000 and later versions of the operating system, **PoSetPowerState** returns the previous power state. On Windows 98/Me, **PoSetPowerState** returns the state passed in *State*.

Remarks

PoSetPowerState notifies the power manager of the new power state for a device. Each driver in a device stack (filter, function, and bus drivers) must call **PoSetPowerState** to inform the power manager of a change in the power state of its corresponding device object. For example:

- A driver calls this routine after receiving a device set-power request and before calling PoStartNextPowerIrp. When handling a PnP IRP_MN_START_DEVICE request, the driver should call **PoSetPowerState** to notify the power manager that the device is in the D0 state.
- If the device is powering down, the driver must call **PoSetPowerState** before leaving the D0 state. In addition, the driver must be able to process client requests before **PoSetPowerState** returns.
- If the device is powering up, the driver must call **PoSetPowerState** after the device is successfully put into the D0 state.

For more information about powering a device up and down, see Handling IRP_MN_SET_POWER for Device Power States.

Callers of **PoSetPowerState** must be running at IRQL <= APC_LEVEL except when setting state to D0. When setting state to D0, callers can be running at IRQL <= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

IRP_MN_START_DEVICE

PoStartNextPowerIrp

PoStartDeviceBusy function

1/25/2019 • 2 minutes to read • Edit Online

The **PoStartDeviceBusy** routine marks the start of a period of time in which the device is busy.

Syntax

```
void PoStartDeviceBusy(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoStartDeviceBusy**.

Return Value

None

Remarks

The **PoStartDeviceBusy** and PoEndDeviceBusy routines mark the start and end of a time period in which a device is busy. Each call to **PoStartDeviceBusy** must be followed by a corresponding call to **PoEndDeviceBusy**.

For each device, the power manager maintains a count of the number of outstanding **PoStartDeviceBusy** calls for which it has not yet received a corresponding **PoEndDeviceBusy** call. A **PoStartDeviceBusy** call increments the busy count by one. A **PoEndDeviceBusy** call decrements the busy count by one. A nonzero busy count disables the idle counter for the device. After the busy count reaches zero, the power manager resets the idle counter to the time-out period and enables the counter.

Rather than use the **PoStartDeviceBusy** and **PoEndDeviceBusy** routines to reset the idle counter, you can call the PoSetDeviceBusyEx routine (or the PoSetDeviceBusy macro). To disable the idle counter during an extended busy period, a driver can call the **PoRegisterDeviceForIdleDetection** routine to disable and enable idle notifications at the start and end of the busy period. However, **PoStartDeviceBusy** and **PoEndDeviceBusy** are typically more convenient to use for this purpose, and you should consider using these routines in new code that you write for Windows 7 and later versions of Windows.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal

Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

PoEndDeviceBusy

PoRegisterDeviceForIdleDetection

PoSetDeviceBusy

PoStartNextPowerIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **PoStartNextPowerIrp** routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)

Syntax

```
void PoStartNextPowerIrp(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to an IRP in which the major function code is IRP_MJ_POWER.

Return Value

None

Remarks

Starting with Windows Vista, the driver is not required to call **PoStartNextPowerIrp** and a call to this routine does not perform a power management operation. However, on Windows Server 2003, Windows XP, and Windows 2000, **PoStartNextPowerIrp** must be called by every driver in a device stack after the driver is finished with the previous power IRP, if any, and is ready to handle the next power IRP. It must be called once by each driver for every IRP_MN_QUERY_POWER or IRP_MN_SET_POWER request.

Although power IRPs are completed only once, typically by the bus driver for a device, each driver in the device stack must call **PoStartNextPowerIrp** as the IRP travels down or back up the stack. Even if a driver fails the IRP, the driver must nevertheless call **PoStartNextPowerIrp** to signal the power manager that it is ready to handle another power IRP.

The driver must call **PoStartNextPowerIrp** while the current IRP stack location points to the current driver. Therefore, this routine must be called before IoCompleteRequest, IoSkipCurrentIrpStackLocation, and PoCallDriver. As a general rule, a driver should call **PoStartNextPowerIrp** from its *IoCompletion* routine associated with the IRP or from the callback routine it passed to PoRequestPowerIrp.

Bus drivers must call **PoStartNextPowerIrp** before completing each IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IRP

IRP_MJ_POWER

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IoCompleteRequest

IoSkipCurrentIrpStackLocation

PoCallDriver

PoRequestPowerIrp

PoUnregisterPowerSettingCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **PoUnregisterPowerSettingCallback** routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.

Syntax

```
NTSTATUS PoUnregisterPowerSettingCallback(
  PVOID Handle
);
```

Parameters

`Handle`

A handle to a callback routine that a driver registered by calling **PoRegisterPowerSettingCallback**.

Return Value

PoUnregisterPowerSettingCallback returns one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The callback routine was unregistered.
STATUS_INVALID_PARAMETER	The *Handle* value is not a valid handle to a power setting callback routine.

Remarks

A driver calls **PoUnregisterPowerSettingCallback** to unregister a power setting callback routine that the driver previously registered by calling **PoRegisterPowerSettingCallback**.

A driver must call **PoUnregisterPowerSettingCallback** to unregister each callback routine that it previously registered. All callback routines registered by a driver should be unregistered in the *Unload* routine of the driver.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

PoRegisterPowerSettingCallback

PoUnregisterSystemState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoUnregisterSystemState** routine cancels a system state registration created by PoRegisterSystemState.

Syntax

```
void PoUnregisterSystemState(
  PVOID StateHandle
);
```

Parameters

`StateHandle`

A pointer to a handle previously returned by **PoRegisterSystemState**.

Return Value

None

Remarks

This routine cancels a system busy state registration established by **PoRegisterSystemState** and frees the associated *StateHandle*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoRegisterSystemState

PsGetCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentThread** routine identifies the current thread.

Syntax

```
PETHREAD PsGetCurrentThread(

);
```

Parameters

This function has no parameters.

Return Value

PsGetCurrentThread returns a pointer to the executive thread object that represents the currently executing thread.

Remarks

This macro

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentThread

PsCreateSystemThread

PsGetCurrentProcess

PsGetCurrentProcessId

PsIsSystemThread

PsIsSystemThread function

1/25/2019 • 2 minutes to read • Edit Online

The **PsIsSystemThread** routine checks whether a given thread is a system thread.

Syntax

```
BOOLEAN PsIsSystemThread(
  PETHREAD Thread
);
```

Parameters

`Thread`

Pointer to the thread to be checked.

Return Value

PsIsSystemThread returns **TRUE** if the specified thread is a system thread, **FALSE** otherwise.

Remarks

For more information about using system threads and managing synchronization within a nonarbitrary thread context, see Driver Threads, Dispatcher Objects, and Resources.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IoIsSystemThread

PsGetCurrentThread

RtlInitStringEx function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInitStringEx** routine initializes a counted string of 8-bit characters.

Syntax

```
NTSYSAPI NTSTATUS RtlInitStringEx(
  PSTRING              DestinationString,
  __drv_aliasesMem PCSZ SourceString
);
```

Parameters

`DestinationString`

A pointer to the **STRING** structure to be initialized. The Ntdef.h header file defines this structure to be identical to the ANSI_STRING structure.

`SourceString`

A pointer to a null-terminated character string. This string is used to initialize the counted string pointed to by *DestinationString*.

Return Value

Returns STATUS_NAME_TOO_LONG if the SourceString is too long. Otherwise, this routine returns STATUS_SUCCESS.

Remarks

This routine initializes a counted character string.

The routine copies the *SourceString* pointer value to the **Buffer** member of the **STRING** structure pointed to by *DestinationString*. The **Length** member of this structure is set to the length, in bytes, of the source string, excluding the terminating null. The **MaximumLength** member of the structure is set to the length, in bytes, of the source string, including the terminating null. If *SourceString* is **NULL**, **Length** and **MaximumLength** are both set to zero.

RtlInitStringEx does not alter the source string pointed to by *SourceString*.

Callers of **RtlInitStringEx** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL = PASSIVE_LEVEL because most other **Rtl*Xxx*String** routines cannot be called at IRQL > PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal

Header	ntifs.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

ANSI_STRING

RtlUnicodeToUTF8N function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeToUTF8N** routine converts a Unicode string to a UTF-8 string.

Syntax

```
NTSYSAPI NTSTATUS RtlUnicodeToUTF8N(
  PCHAR  UTF8StringDestination,
  ULONG  UTF8StringMaxByteCount,
  PULONG UTF8StringActualByteCount,
  PCWCH  UnicodeStringSource,
  ULONG  UnicodeStringByteCount
);
```

Parameters

`UTF8StringDestination`

A pointer to a caller-allocated destination buffer into which the routine writes the UTF-8 output string. If this parameter is **NULL**, the routine writes the required size of the output buffer to **UTF8StringActualByteCount*.

`UTF8StringMaxByteCount`

Specifies the maximum number of bytes that the routine can write to the buffer that *UTF8StringDestination* points to. If *UTF8StringDestination* = **NULL**, set *UTF8StringMaxByteCount* = 0.

`UTF8StringActualByteCount`

A pointer to a location into which the routine writes the actual number of bytes that it has written to the buffer that *UTF8StringDestination* points to. If *UTF8StringDestination* is non-**NULL**, this count never exceeds the value of *UTF8StringMaxByteCount*. If *UTF8StringDestination* is **NULL**, this count is the number of bytes that are required to contain the entire output string.

`UnicodeStringSource`

A pointer to the Unicode source string.

`UnicodeStringByteCount`

Specifies the number of bytes in the Unicode source string that the *UnicodeStringSource* parameter points to.

Return Value

RtlUnicodeToUTF8N returns STATUS_SUCCESS if the call is successful and all Unicode character codes in the input string were converted to the corresponding UTF-8 character codes in the output string. It returns STATUS_SOME_NOT_MAPPED if the call is successful but one or more input characters were invalid and were replaced by the Unicode replacement character, U+FFFD, before they were converted to UTF-8. Possible error return values include the following error codes:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_TOO_SMALL	The *UTF8StringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string.
STATUS_INVALID_PARAMETER	The *UTF8StringDestination* and *UTF8StringActualByteCount* parameters are both **NULL**.
STATUS_INVALID_PARAMETER_4	The *UnicodeStringSource* parameter is **NULL**.
STATUS_INVALID_PARAMETER_5	*UnicodeStringByteCount* is not an integer multiple of **sizeof**(WCHAR).

Remarks

The UTF-8 output string is null-terminated only if the Unicode input string is null-terminated.

The routine returns STATUS_BUFFER_TOO_SMALL if the *UTF8StringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string. In this case, the routine writes as many UTF-8 characters as will fit in the buffer, and the **UTF8StringActualByteCount* value specifies the number of valid bytes that the routine has written to the buffer. The partial string that is contained in the output buffer might not include a terminating null character.

You can make an initial call to **RtlUnicodeToUTF8N** to obtain the required output buffer size, and then call **RtlUnicodeToUTF8N** again to obtain the Unicode output string. In the initial call, set *UTF8StringDestination* = **NULL** and *UTF8StringMaxByteCount* = 0, and the routine will write the required buffer size to **UTF8StringActualByteCount*. Next, allocate a buffer of the required size and call **RtlUnicodeToUTF8N** a second time to obtain the UTF-8 output string.

RtlUnicodeToUTF8N supports Unicode surrogate pairs. However, a surrogate leading word value that is not followed by a trailing word value, or a trailing word value that is not preceded by a leading word value, is not recognized as a valid character code and is replaced by the Unicode replacement character, U+FFFD.

RtlUnicodeToUTF8N continues to convert the input string to an output string until it reaches the end of the source buffer or the end of the destination buffer, whichever occurs first. The routine converts any null characters in the input string to null characters in the output string. If the input string contains a terminating null character, but the null character is not located at the end of the source buffer, the routine continues past the terminating null character until it reaches the end of the available buffer space.

The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.

You can use **RtlUnicodeToUTF8N** and **RtlUTF8ToUnicode** routines to perform a lossless conversion of valid text strings between the Unicode and UTF-8 formats. However, strings that have arbitrary data values are likely to violate the Unicode rules for encoding surrogate pairs, and any information that is contained in the invalid values in an input string is lost and cannot be recovered from the resulting output string.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlUTF8ToUnicodeN

RtlUTF8ToUnicodeN function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUTF8ToUnicodeN** routine converts a UTF-8 string to a Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlUTF8ToUnicodeN(
  PWSTR  UnicodeStringDestination,
  ULONG  UnicodeStringMaxByteCount,
  PULONG UnicodeStringActualByteCount,
  PCCH   UTF8StringSource,
  ULONG  UTF8StringByteCount
);
```

Parameters

`UnicodeStringDestination`

A pointer to a caller-allocated destination buffer into which the routine writes the Unicode output string. If this parameter is **NULL**, the routine writes the required size of the output buffer to **UnicodeStringActualByteCount*.

`UnicodeStringMaxByteCount`

Specifies the maximum number of bytes that the routine can write to the buffer that *UnicodeStringDestination* points to. If *UnicodeStringDestination* = **NULL**, set *UnicodeStringMaxByteCount* = 0.

`UnicodeStringActualByteCount`

A pointer to a location into which the routine writes the actual number of bytes that the routine has written to the buffer that *UnicodeStringDestination* points to. If *UnicodeStringDestination* is non-**NULL**, this count never exceeds the value of *UnicodeStringMaxByteCount*. If *UnicodeStringDestination* is **NULL**, this count is the number of bytes that are required to contain the entire output string.

`UTF8StringSource`

A pointer to the UTF-8 source string.

`UTF8StringByteCount`

Specifies the number of bytes in the UTF-8 source string that the *UTF8StringSource* parameter points to.

Return Value

RtlUTF8ToUnicodeN returns STATUS_SUCCESS if the call is successful and all UTF-8 character codes in the input string were converted to the corresponding Unicode character codes in the output string. It returns STATUS_SOME_NOT_MAPPED if the call is successful but one or more input characters were invalid and were converted to the Unicode replacement character, U+FFFD. Possible error return values include the following error codes:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_TOO_SMALL	The *UnicodeStringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string.
STATUS_INVALID_PARAMETER	The *UnicodeStringDestination* and *UnicodeStringActualByteCount* parameters are both **NULL**.
STATUS_INVALID_PARAMETER_4	The *UTF8StringSource* parameter is **NULL**.

Remarks

The Unicode output string is null-terminated only if the UTF-8 input string is null-terminated.

The routine returns STATUS_BUFFER_TOO_SMALL if the *UnicodeStringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string. In this case, the routine writes as many Unicode characters as will fit in the buffer, and the **UnicodeStringActualByteCount* value specifies the number of valid bytes that the routine has written to the buffer. The partial string that is contained in the output buffer might not include a terminating null character.

You can make an initial call to **RtlUTF8ToUnicodeN** to obtain the required output buffer size, and then call **RtlUTF8ToUnicodeN** again to obtain the Unicode output string. In the initial call, set *UnicodeStringDestination* = **NULL** and *UnicodeStringMaxByteCount* = 0, and the routine will write the required buffer size to **UnicodeStringActualByteCount*. Next, allocate a buffer of the required size and call **RtlUTF8ToUnicodeN** a second time to obtain the Unicode output string.

RtlUTF8ToUnicodeN supports Unicode surrogate pairs. However, a surrogate leading word value that is not followed by a trailing word value, or a trailing word value that is not preceded by a leading word value, is not recognized as a valid character and is replaced by the Unicode replacement character, U+FFFD.

RtlUTF8ToUnicodeN continues to convert the input string to an output string until it reaches the end of the source buffer or the end of the destination buffer, whichever occurs first. The routine converts any null characters in the input string to null characters in the output string. If the input string contains a terminating null character, but the null character is not located at the end of the source buffer, the routine continues past the terminating null character until it reaches the end of the available buffer space.

The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.

You can use the **RtlUTF8ToUnicode** and **RtlUnicodeToUTF8N** routines to perform a lossless conversion of valid text strings between the UTF-8 and Unicode formats. However, strings that have arbitrary data values are likely to violate the Unicode rules for encoding surrogate pairs, and any information that is contained in the invalid values in an input string is lost and cannot be recovered from the resulting output string.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	ntifs.h (include Ntifs.h, Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeToUTF8N

SeFreePrivileges function

1/25/2019 • 2 minutes to read • Edit Online

The **SeFreePrivileges** routine frees a privilege set returned by **SeAccessCheck**.

Syntax

```
void SeFreePrivileges(
  PPRIVILEGE_SET Privileges
);
```

Parameters

`Privileges`

Pointer to the privilege set to be freed.

Return Value

None

Remarks

For more information about security and access control, see the documentation on these topics in the Microsoft Windows SDK.

Requirements

Target Platform	Universal
Header	ntifs.h (include Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

PRIVILEGE_SET

SeAccessCheck

SeAppendPrivileges

SePrivilegeCheck

ntimage.h header

1/8/2019 • 2 minutes to read • Edit Online

This header file that describes all image structures. For more information, see IMAGE_* structures under Winnt.h

ntintsafe.h header

1/8/2019 • 12 minutes to read • Edit Online

This header is used by kernel. For more information, see:

- Windows kernel ntintsafe.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
RtlByteToChar	Converts a value of type BYTE to a value of type CHAR.
RtlByteToInt8	Converts a value of type BYTE to a value of type INT8.
RtlDWordPtrAdd	Adds two values of type DWORD_PTR.
RtlDWordPtrMult	Multiplies one value of type DWORD_PTR by another.
RtlDWordPtrSub	Subtracts one value of type DWORD_PTR from another.
RtlInt8Add	Adds two values of type INT8.
RtlInt8Mult	Multiplies one value of type INT8 by another.
RtlInt8Sub	Subtracts one value of type INT8 from another.
RtlInt8ToUChar	Converts a value of type INT8 to a value of type UCHAR.
RtlInt8ToUInt	Converts a value of type INT8 to a value of type UINT.
RtlInt8ToUInt8	Converts a value of type INT8 to a value of type UINT8.
RtlInt8ToUIntPtr	Converts a value of type INT8 to a value of type UINT_PTR.
RtlInt8ToULong	Converts a value of type INT8 to a value of type ULONG.
RtlInt8ToULongLong	Converts a value of type INT8 to a value of type ULONGLONG.
RtlInt8ToULongPtr	Converts a value of type INT8 to a value of type ULONG_PTR.
RtlInt8ToUShort	Converts a value of type INT8 to a value of type USHORT.
RtlIntAdd	Adds two values of type INT.
RtlIntMult	Multiplies one value of type INT by another.
RtlIntPtrAdd	Adds two values of type INT_PTR.

TITLE	DESCRIPTION
RtlIntPtrMult	Multiplies one value of type INT_PTR by another.
RtlIntPtrSub	Subtracts one value of type INT_PTR from another.
RtlIntPtrToChar	Converts a value of type INT_PTR to a value of type CHAR.
RtlIntPtrToInt	Converts a value of type INT_PTR to a value of type INT.
RtlIntPtrToInt8	Converts a value of type INT_PTR to a value of type INT8.
RtlIntPtrToLong	Converts a value of type INT_PTR to a value of type LONG.
RtlIntPtrToLongPtr	Converts a value of type INT_PTR to a value of type LONG_PTR.
RtlIntPtrToShort	Converts a value of type INT_PTR to a value of type SHORT.
RtlIntPtrToUChar	Converts a value of type INT_PTR to a value of type UCHAR.
RtlIntPtrToUInt	Converts a value of type INT_PTR to a value of type UINT.
RtlIntPtrToUInt8	Converts a value of type INT_PTR to a value of type UINT8.
RtlIntPtrToUIntPtr	Converts a value of type INT_PTR to a value of type UINT_PTR.
RtlIntPtrToULong	Converts a value of type INT_PTR to a value of type ULONG.
RtlIntPtrToULongLong	Converts a value of type INT_PTR to a value of type ULONGLONG.
RtlIntPtrToULongPtr	Converts a value of type INT_PTR to a value of type ULONG_PTR.
RtlIntPtrToUShort	Converts a value of type INT_PTR to a value of type USHORT.
RtlIntSub	Subtracts one value of type INT from another.
RtlIntToChar	Converts a value of type INT to a value of type CHAR.
RtlIntToInt8	Converts a value of type INT to a value of type INT8.
RtlIntToShort	Converts a value of type INT to a value of type SHORT.
RtlIntToUChar	Converts a value of type INT to a value of type UCHAR.
RtlIntToUInt	Converts a value of type INT to a value of type UINT.
RtlIntToUInt8	Converts a value of type INT to a value of type UINT8.
RtlIntToULong	Converts a value of type INT to a value of type ULONG.

TITLE	DESCRIPTION
RtlIntToULongLong	Converts a value of type INT to a value of type ULONGLONG.
RtlIntToUShort	Converts a value of type INT to a value of type USHORT.
RtlLongAdd	Adds two values of type LONG.
RtlLongLongAdd	Adds two values of type LONGLONG.
RtlLongLongMult	Multiplies one value of type LONGLONG by another.
RtlLongLongSub	Subtracts one value of type LONGLONG from another.
RtlLongLongToChar	Converts a value of type LONGLONG to a value of type CHAR.
RtlLongLongToInt	Converts a value of type LONGLONG to a value of type INT.
RtlLongLongToInt8	Converts a value of type LONGLONG to a value of type INT8.
RtlLongLongToIntPtr	Converts a value of type LONGLONG to a value of type INT_PTR.
RtlLongLongToLong	Converts a value of type LONGLONG to a value of type LONG.
RtlLongLongToLongPtr	Converts a value of type LONGLONG to a value of type LONG_PTR.
RtlLongLongToShort	Converts a value of type LONGLONG to a value of type SHORT.
RtlLongLongToUChar	Converts a value of type LONGLONG to a value of type UCHAR.
RtlLongLongToUInt	Converts a value of type LONGLONG to a value of type UINT.
RtlLongLongToUInt8	Converts a value of type LONGLONG to a value of type UNIT8.
RtlLongLongToULong	Converts a value of type LONGLONG to a value of type ULONG.
RtlLongLongToULongLong	Converts a value of type LONGLONG to a value of type LONGLONG.
RtlLongLongToUShort	Converts a value of type LONGLONG to a value of type USHORT.
RtlLongMult	Multiplies one value of type LONG by another.
RtlLongPtrAdd	Adds two values of type LONG_PTR.

TITLE	DESCRIPTION
RtlLongPtrMult	Multiplies one value of type LONG_PTR by another.
RtlLongPtrSub	Subtracts one value of type LONG_PTR from another.
RtlLongPtrToChar	Converts a value of type LONG_PTR to a value of type CHAR.
RtlLongPtrToInt	Converts a value of type LONG_PTR to a value of type INT.
RtlLongPtrToInt8	Converts a value of type LONG_PTR to a value of type INT8.
RtlLongPtrToIntPtr	Converts a value of type LONG_PTR to a value of type INT_PTR.
RtlLongPtrToLong	Converts a value of type LONG_PTR to a value of type LONG.
RtlLongPtrToShort	Converts a value of type LONG_PTR to a value of type SHORT.
RtlLongPtrToUChar	Converts a value of type LONG_PTR to a value of type UCHAR.
RtlLongPtrToUInt	Converts a value of type LONG_PTR to a value of type UINT.
RtlLongPtrToUInt8	Converts a value of type LONG_PTR to a value of type UINT8.
RtlLongPtrToUIntPtr	Converts a value of type LONG_PTR to a value of type UINT_PTR.
RtlLongPtrToULong	Converts a value of type LONG_PTR to a value of type ULONG.
RtlLongPtrToULongLong	Converts a value of type LONG_PTR to a value of type ULONGLONG.
RtlLongPtrToULongPtr	Converts a value of type LONG_PTR to a value of type ULONG_PTR.
RtlLongPtrToUShort	Converts a value of type LONG_PTR to a value of type USHORT.
RtlLongSub	Subtracts one value of type LONG from another.
RtlLongToChar	Converts a value of type LONG to a value of type CHAR.
RtlLongToInt	Converts a value of type LONG to a value of type INT.
RtlLongToInt8	Converts a value of type LONG to a value of type INT8.
RtlLongToIntPtr	Converts a value of type LONG to a value of type INT_PTR.
RtlLongToShort	Converts a value of type LONG to a value of type SHORT.

TITLE	DESCRIPTION
RtlLongToUChar	Converts a value of type LONG to a value of type UCHAR.
RtlLongToUInt	Converts a value of type LONG to a value of type UINT.
RtlLongToUInt8	Converts a value of type LONG to a value of type UINT8.
RtlLongToUIntPtr	Converts a value of type LONG to a value of type UINT_PTR.
RtlLongToULong	Converts a value of type LONG to a value of type ULONG.
RtlLongToULongLong	Converts a value of type LONG to a value of type ULONGLONG.
RtlLongToULongPtr	Converts a value of type LONG to a value of type ULONG_PTR.
RtlLongToUShort	Converts a value of type LONG to a value of type USHORT.
RtlPtrdiffTAdd	Adds two values of type PTRDIFF_T.
RtlPtrdiffTMult	Multiplies one value of type PTRDIFF_T by another.
RtlPtrdiffTSub	Subtracts one value of type PTRDIFF_T from another.
RtlShortAdd	Adds two values of type SHORT.
RtlShortMult	Multiplies one value of type SHORT by another.
RtlShortSub	Subtracts one value of type SHORT from another.
RtlShortToChar	Converts a value of type SHORT to a value of type CHAR.
RtlShortToDWordPtr	Converts a value of type SHORT to a value of type DWORD_PTR.
RtlShortToInt8	Converts a value of type SHORT to a value of type INT8.
RtlShortToUChar	Converts a value of type SHORT to a value of type UCHAR.
RtlShortToUInt	Converts a value of type SHORT to a value of type UINT.
RtlShortToUInt8	Converts a value of type SHORT to a value of type UINT8.
RtlShortToUIntPtr	Converts a value of type SHORT to a value of type UINT_PTR.
RtlShortToULong	Converts a value of type SHORT to a value of type ULONG.
RtlShortToULongLong	Converts a value of type SHORT to a value of type ULONGLONG.

TITLE	DESCRIPTION
RtlShortToULongPtr	Converts a value of type SHORT to a value of type ULONG_PTR.
RtlShortToUShort	Converts a value of type SHORT to a value of type USHORT.
RtlSizeTAdd	Adds two values of type SIZE_T.
RtlSizeTMult	Multiplies one value of type SIZE_T by another.
RtlSizeTSub	Subtracts one value of type SIZE_T from another.
RtlSSIZETAdd	Adds two values of type SSIZE_T.
RtlSSIZETMult	Multiplies one value of type SSIZE_T by another.
RtlSSIZETSub	Subtracts one value of type SSIZE_T from another.
RtlUInt8Add	Adds two values of type UINT8.
RtlUInt8Mult	Multiplies one value of type UINT8 by another.
RtlUInt8Sub	The RtlUInt8Sub routine subtracts one value of type UINT8 from another.
RtlUInt8ToChar	Converts a value of type UINT8 to a value of type CHAR.
RtlUInt8ToInt8	Converts a value of type UINT8 to a value of type INT8.
RtlUIntAdd	Adds two values of type UINT.
RtlUIntMult	Multiplies one value of type UINT by another.
RtlUIntPtrAdd	Adds two values of type UINT_PTR.
RtlUIntPtrMult	Multiplies one value of type UINT_PTR by another.
RtlUIntPtrSub	Subtracts one value of type UINT_PTR from another.
RtlUIntPtrToChar	Converts a value of type UINT_PTR to a value of type CHAR.
RtlUIntPtrToInt	Converts a value of type UINT_PTR to a value of type INT.
RtlUIntPtrToInt16	Converts a value of type UINT_PTR to a value of type INT16.
RtlUIntPtrToInt8	Converts a value of type UINT_PTR to a value of type INT8.
RtlUIntPtrToIntPtr	Converts a value of type UINT_PTR to a value of type INT_PTR.
RtlUIntPtrToLong	Converts a value of type UINT_PTR to a value of type LONG.

TITLE	DESCRIPTION
RtlUIntPtrToLongLong	Converts a value of type UINT_PTR to a value of type LONGLONG.
RtlUIntPtrToLongPtr	Converts a value of type UINT_PTR to a value of type LONG_PTR.
RtlUIntPtrToShort	Converts a value of type UINT_PTR to a value of type SHORT.
RtlUIntPtrToUChar	Converts a value of type UINT_PTR to a value of type UCHAR.
RtlUIntPtrToUInt	Converts a value of type UINT_PTR to a value of type UINT.
RtlUIntPtrToUInt16	Converts a value of type UINT_PTR to a value of type UINT16.
RtlUIntPtrToUInt8	Converts a value of type UINT_PTR to a value of type UINT8.
RtlUIntPtrToULong	Converts a value of type UINT_PTR to a value of type LONG.
RtlUIntPtrToUShort	Converts a value of type UINT_PTR to a value of type USHORT.
RtlUIntSub	Subtracts one value of type UINT from another.
RtlUIntToChar	Converts a value of type UINT to a value of type CHAR.
RtlUIntToInt	Converts a value of type UINT to a value of type INT.
RtlUIntToInt8	Converts a value of type UINT to a value of type INT8.
RtlUIntToIntPtr	Converts a value of type UINT to a value of type INT_PTR.
RtlUIntToLong	Converts a value of type UINT to a value of type LONG.
RtlUIntToLongPtr	Converts a value of type UINT to a value of type LONG_PTR.
RtlUIntToShort	Converts a value of type UINT to a value of type SHORT.
RtlUIntToUChar	Converts a value of type UINT to a value of type UCHAR.
RtlUIntToUInt8	Converts a value of type UINT to a value of type UINT8.
RtlUIntToUShort	Converts a value of type UINT to a value of type USHORT.
RtlULongAdd	Adds two values of type ULONG.
RtlULongLongAdd	Adds two values of type ULONGLONG.
RtlULongLongMult	Multiplies one value of type ULONGLONG by another.
RtlULongLongSub	Subtracts one value of type ULONGLONG from another.

TITLE	DESCRIPTION
RtlULongLongToChar	Converts a value of type ULONGLONG to a value of type CHAR.
RtlULongLongToInt	Converts a value of type ULONGLONG to a value of type INT.
RtlULongLongToInt8	Converts a value of type ULONGLONG to a value of type INT8.
RtlULongLongToLong	Converts a value of type ULONGLONG to a value of type LONG.
RtlULongLongToLongLong	Converts a value of type ULONGLONG to a value of type LONGLONG.
RtlULongLongToLongPtr	Converts a value of type ULONGLONG to a value of type LONG_PTR.
RtlULongLongToShort	Converts a value of type ULONGLONG to a value of type SHORT.
RtlULongLongToUChar	Converts a value of type ULONGLONG to a value of type UCHAR.
RtlULongLongToUInt	Converts a value of type ULONGLONG to a value of type UINT.
RtlULongLongToUInt8	Converts a value of type ULONGLONG to a value of type UINT8.
RtlULongLongToUIntPtr	Converts a value of type ULONGLONG to a value of type UINT_PTR.
RtlULongLongToULong	Converts a value of type ULONGLONG to a value of type ULONG.
RtlULongLongToULongPtr	Converts a value of type ULONGLONG to a value of type ULONG_PTR.
RtlULongLongToUShort	Converts a value of type ULONGLONG to a value of type USHORT.
RtlULongMult	Multiplies one value of type ULONG by another.
RtlULongPtrAdd	Adds two values of type ULONG_PTR.
RtlULongPtrMult	Multiplies one value of type ULONG_PTR by another.
RtlULongPtrSub	Subtracts one value of type ULONG_PTR from another.
RtlULongPtrToChar	Converts a value of type ULONG_PTR to a value of type CHAR.
RtlULongPtrToInt	Converts a value of type ULONG_PTR to a value of type INT.

TITLE	DESCRIPTION
RtlULongPtrToInt8	Converts a value of type ULONG_PTR to a value of type INT8.
RtlULongPtrToIntPtr	Converts a value of type ULONG_PTR to a value of type INT_PTR.
RtlULongPtrToLong	Converts a value of type ULONG_PTR to a value of type LONG.
RtlULongPtrToLongLong	Converts a value of type ULONG_PTR to a value of type LONGLONG.
RtlULongPtrToLongPtr	Converts a value of type ULONG_PTR to a value of type LONG_PTR.
RtlULongPtrToShort	Converts a value of type ULONG_PTR to a value of type SHORT.
RtlULongPtrToUChar	Converts a value of type ULONG_PTR to a value of type UCHAR.
RtlULongPtrToUInt	Converts a value of type ULONG_PTR to a value of type UINT.
RtlULongPtrToUInt8	Converts a value of type ULONG_PTR to a value of type UINT8.
RtlULongPtrToUIntPtr	Converts a value of type ULONG_PTR to a value of type UINT_PTR.
RtlULongPtrToULong	Converts a value of type ULONG_PTR to a value of type ULONG.
RtlULongPtrToUShort	Converts a value of type ULONG_PTR to a value of type USHORT.
RtlULongSub	Subtracts one value of type ULONG from another.
RtlULongToChar	Converts a value of type ULONG to a value of type CHAR.
RtlULongToInt	Converts a value of type ULONG to a value of type INT.
RtlULongToInt8	Converts a value of type ULONG to a value of type INT8.
RtlULongToIntPtr	Converts a value of type ULONG to a value of type INT_PTR.
RtlULongToLong	Converts a value of type ULONG to a value of type LONG.
RtlULongToLongPtr	Converts a value of type ULONG to a value of type LONG_PTR.
RtlULongToShort	Converts a value of type ULONG to a value of type SHORT.
RtlULongToUChar	Converts a value of type ULONG to a value of type UCHAR.

TITLE	DESCRIPTION
RtlULongToUInt	Converts a value of type ULONG to a value of type UINT.
RtlULongToUInt8	Converts a value of type ULONG_PTR to a value of type UINT8.
RtlULongToUIntPtr	Converts a value of type ULONG_PTR to a value of type UINT_PTR.
RtlULongToUShort	Converts a value of type ULONG to a value of type USHORT.
RtlUShortAdd	Adds two values of type USHORT.
RtlUShortMult	Multiplies one value of type USHORT by another.
RtlUShortSub	Subtracts one value of type USHORT from another.
RtlUShortToChar	Converts a value of type USHORT to a value of type CHAR.
RtlUShortToInt8	Converts a value of type USHORT to a value of type INT8.
RtlUShortToShort	Converts a value of type USHORT to a value of type SHORT.
RtlUShortToUChar	Converts a value of type USHORT to a value of type UCHAR.
RtlUShortToUInt8	Converts a value of type USHORT to a value of type UINT8.

RtlByteToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **BYTE** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlByteToChar(
  BYTE bOperand,
  CHAR *pch
);
```

Parameters

`bOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlByteToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **BYTE** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlByteToInt8(
  BYTE bOperand,
  INT8 *pi8Result
);
```

Parameters

`bOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlDWordPtrAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **DWORD_PTR**.

Syntax

```
NTSTATUS RtlDWordPtrAdd(
  DWORD_PTR dwAugend,
  DWORD_PTR dwAddend,
  DWORD_PTR *pdwResult
);
```

Parameters

`dwAugend`

The first value in the equation.

`dwAddend`

The value to add to *dwAugend*.

`pdwResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlDWordPtrMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **DWORD_PTR** by another.

Syntax

```
NTSTATUS RtlDWordPtrMult(
  DWORD_PTR dwMultiplicand,
  DWORD_PTR dwMultiplier,
  DWORD_PTR *pdwResult
);
```

Parameters

`dwMultiplicand`

The value to be multiplied by *dwMultiplier*.

`dwMultiplier`

The value by which to multiply *dwMultiplicand*.

`pdwResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlDWordPtrSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **DWORD_PTR** from another.

Syntax

```
NTSTATUS RtlDWordPtrSub(
  DWORD_PTR dwMinuend,
  DWORD_PTR dwSubtrahend,
  DWORD_PTR *pdwResult
);
```

Parameters

`dwMinuend`

The value from which *dwSubtrahend* is subtracted.

`dwSubtrahend`

The value to subtract from *dwMinuend*.

`pdwResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8Add function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **INT8**.

Syntax

```
NTSTATUS RtlInt8Add(
  INT8 i8Augend,
  INT8 i8Addend,
  INT8 *pi8Result
);
```

Parameters

`i8Augend`

The first value in the equation.

`i8Addend`

The value to add to *i8Augend*.

`pi8Result`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8Mult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **INT8** by another.

Syntax

```
NTSTATUS RtlInt8Mult(
  INT8 i8Multiplicand,
  INT8 i8Multiplier,
  INT8 *pi8Result
);
```

Parameters

`i8Multiplicand`

The value to be multiplied by *i8Multiplier*.

`i8Multiplier`

The value by which to multiply *i8Multiplicand*.

`pi8Result`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8Sub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **INT8** from another.

Syntax

```
NTSTATUS RtlInt8Sub(
  INT8 i8Minuend,
  INT8 i8Subtrahend,
  INT8 *pi8Result
);
```

Parameters

`i8Minuend`

The value from which *i8Subtrahend* is subtracted.

`i8Subtrahend`

The value to subtract from *i8Minuend*.

`pi8Result`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlInt8ToUChar(
  INT8  i8Operand,
  UCHAR *pch
);
```

Parameters

`i8Operand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlInt8ToUInt(
  INT8 i8Operand,
  UINT *puResult
);
```

Parameters

`i8Operand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToUInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlInt8ToUInt8(
  INT8  i8Operand,
  UINT8 *pu8Result
);
```

Parameters

`i8Operand`

The value to be converted.

`pu8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlInt8ToUIntPtr(
  INT8     i8Operand,
  UINT_PTR *puResult
);
```

Parameters

`i8Operand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlInt8ToULong(
  INT8  i8Operand,
  ULONG *pulResult
);
```

Parameters

`i8Operand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlInt8ToULongLong(
  INT8      i8Operand,
  ULONGLONG *pullResult
);
```

Parameters

`i8Operand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToDWordLong
- RtlInt8ToULong64
- RtlInt8ToDWord64
- RtlInt8ToUInt64

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlInt8ToULongPtr(
  INT8      i8Operand,
  ULONG_PTR *pulResult
);
```

Parameters

`i8Operand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToDWordPtr
- RtlInt8ToSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlInt8ToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT8** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlInt8ToUShort(
  INT8   i8Operand,
  USHORT *pusResult
);
```

Parameters

`i8Operand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt8ToUInt16
- RtlInt8ToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **INT**.

Syntax

```
NTSTATUS RtlIntAdd(
  INT iAugend,
  INT iAddend,
  INT *piResult
);
```

Parameters

`iAugend`

The first value in the equation.

`iAddend`

The value to add to *iAugend*.

`piResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32Add
- RtlLong32Add

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **INT** by another.

Syntax

```
NTSTATUS RtlIntMult(
  INT iMultiplicand,
  INT iMultiplier,
  INT *piResult
);
```

Parameters

`iMultiplicand`

The value to be multiplied by *iMultiplier*.

`iMultiplier`

The value by which to multiply *iMultiplicand*.

`piResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32Mult
- RtlLong32Mult

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **INT_PTR**.

Syntax

```
NTSTATUS RtlIntPtrAdd(
  INT_PTR iAugend,
  INT_PTR iAddend,
  INT_PTR *piResult
);
```

Parameters

`iAugend`

The first value in the equation.

`iAddend`

The value to add to *iAugend*.

`piResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **INT_PTR** by another.

Syntax

```
NTSTATUS RtlIntPtrMult(
  INT_PTR iMultiplicand,
  INT_PTR iMultiplier,
  INT_PTR *piResult
);
```

Parameters

`iMultiplicand`

The value to be multiplied by *iMultiplier*.

`iMultiplier`

The value by which to multiply *iMultiplicand*.

`piResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **INT_PTR** from another.

Syntax

```
NTSTATUS RtlIntPtrSub(
  INT_PTR iMinuend,
  INT_PTR iSubtrahend,
  INT_PTR *piResult
);
```

Parameters

`iMinuend`

The value from which *iSubtrahend* is subtracted.

`iSubtrahend`

The value to subtract from *iMinuend*.

`piResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlIntPtrToChar(
  INT_PTR iOperand,
  CHAR    *pch
);
```

Parameters

`iOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlPtrdiffTToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **INT**.

Syntax

```
NTSTATUS RtlIntPtrToInt(
  INT_PTR iOperand,
  INT     *piResult
);
```

Parameters

`iOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToInt32
- RtlPtrdiffTToInt
- RtlPtrdiffTToInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlIntPtrToInt8(
  INT_PTR iOperand,
  INT8    *pi8Result
);
```

Parameters

`iOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlPtrdiffTToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlIntPtrToLong(
  INT_PTR iOperand,
  LONG    *plResult
);
```

Parameters

`iOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlPtrdiffTToLong

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlIntPtrToLongPtr(
  INT_PTR  iOperand,
  LONG_PTR *plResult
);
```

Parameters

`iOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlPtrdiffTToLongPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlIntPtrToShort(
  INT_PTR iOperand,
  SHORT   *psResult
);
```

Parameters

`iOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToInt16
- RtlPtrdiffTToShort
- RtlPtrdiffTToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlIntPtrToUChar(
  INT_PTR iOperand,
  UCHAR   *pch
);
```

Parameters

`iOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlPtrdiffTToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlIntPtrToUInt(
  INT_PTR iOperand,
  UINT    *puResult
);
```

Parameters

`iOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToUInt32
- RtlPtrdiffTToUInt
- RtlPtrdiffTToUInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlIntPtrToUInt8(
  INT_PTR iOperand,
  UINT8   *pui8Result
);
```

Parameters

`iOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToByte
- RtlPtrdiffTToUInt8
- RtlPtrdiffTToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlIntPtrToUIntPtr(
  INT_PTR  iOperand,
  UINT_PTR *puResult
);
```

Parameters

`iOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToSizeT
- RtlPtrdiffTToUIntPtr
- RtlPtrdiffTToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlIntPtrToULong(
  INT_PTR iOperand,
  ULONG   *pulResult
);
```

Parameters

`iOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToDWord
- RtlPtrdiffTToULong
- RtlPtrdiffTToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlIntPtrToULongLong(
  INT_PTR   iOperand,
  ULONGLONG *pullResult
);
```

Parameters

`iOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToDWordLong
- RtlIntPtrToULong64
- RtlIntPtrToDWord64
- RtlIntPtrToUInt64
- RtlPtrdiffTToULongLong
- RtlPtrdiffTToDWordLong
- RtlPtrdiffTToULong64
- RtlPtrdiffTToDWord64
- RtlPtrdiffTToUInt64

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlIntPtrToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlIntPtrToULongPtr(
  INT_PTR   iOperand,
  ULONG_PTR *pulResult
);
```

Parameters

`iOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToDWordPtr
- RtlIntPtrToSIZET
- RtlPtrdiffTToULongPtr
- RtlPtrdiffTToDWordPtr
- RtlPtrdiffTToSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntPtrToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT_PTR** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlIntPtrToUShort(
  INT_PTR iOperand,
  USHORT  *pusResult
);
```

Parameters

`iOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToUInt16
- RtlIntPtrToWord
- RtlPtrdiffTToUShort
- RtlPtrdiffTToUInt16
- RtlPtrdiffTToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **INT** from another.

Syntax

```
NTSTATUS RtlIntSub(
  INT iMinuend,
  INT iSubtrahend,
  INT *piResult
);
```

Parameters

`iMinuend`

The value from which *iSubtrahend* is subtracted.

`iSubtrahend`

The value to subtract from *iMinuend*.

`piResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32Sub
- RtlLong32Sub

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlIntToChar(
  INT  iOperand,
  CHAR *pch
);
```

Parameters

`iOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32ToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlIntToInt8(
  INT  iOperand,
  INT8 *pi8Result
);
```

Parameters

`iOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32ToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlIntToShort(
  INT   iOperand,
  SHORT *psResult
);
```

Parameters

`iOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToInt16
- RtlInt32ToShort
- RtlInt32ToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlIntToUChar(
  INT   iOperand,
  UCHAR *pch
);
```

Parameters

`iOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt32ToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlIntToUInt(
  INT  iOperand,
  UINT *puResult
);
```

Parameters

`iOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToUIntPtr
- RtlInt32ToUInt
- RtlInt32ToUInt32
- RtlIntToSizeT
- RtlInt32ToUIntPtr
- RtlInt32ToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlIntToUInt8(
  INT   iOperand,
  UINT8 *pui8Result
);
```

Parameters

`iOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToByte
- RtlInt32ToByte
- RtlInt32ToUInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlIntToULong(
  INT   iOperand,
  ULONG *pulResult
);
```

Parameters

`iOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToULongPtr
- RtlIntToDWord
- RtlInt32ToULong
- RtlInt32ToDWord
- RtlIntToDWordPtr
- RtlIntToSIZET
- RtlInt32ToULongPtr
- RtlInt32ToDWordPtr
- RtlInt32ToSIZET
- RtlIntToULongPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlIntToULongLong(
  INT       iOperand,
  ULONGLONG *pullResult
);
```

Parameters

`iOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToDWordLong
- RtlIntToULong64
- RtlIntToDWord64
- RtlIntToUInt64
- RtlInt32ToULongLong
- RtlInt32ToDWordLong
- RtlInt32ToULong64
- RtlInt32ToDWord64
- RtlInt32ToUInt64
- RtlIntToUIntPtr
- RtlIntToSizeT
- RtlInt32ToUIntPtr
- RtlInt32ToSizeT
- RtlIntToUIntPtr
- RtlIntToULongPtr

- RtlIntToDWordPtr
- RtlIntToSIZET
- RtlInt32ToULongPtr
- RtlInt32ToDWordPtr
- RtlInt32ToSIZET
- RtlIntToULongPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlIntToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **INT** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlIntToUShort(
  INT    iOperand,
  USHORT *pusResult
);
```

Parameters

`iOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntToUInt16
- RtlIntToWord
- RtlInt32ToUShort
- RtlInt32ToUInt16
- RtlInt32ToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **LONG**.

Syntax

```
NTSTATUS RtlLongAdd(
  LONG lAugend,
  LONG lAddend,
  LONG *plResult
);
```

Parameters

`lAugend`

The first value in the equation.

`lAddend`

The value to add to *lAugend*.

`plResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **LONGLONG**.

Syntax

```
NTSTATUS RtlLongLongAdd(
  LONGLONG llAugend,
  LONGLONG llAddend,
  LONGLONG *pllResult
);
```

Parameters

`llAugend`

The first value in the equation.

`llAddend`

The value to add to *llAugend*.

`pllResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrAdd
- RtlLongPtrAdd
- RtlLong64Add
- RtlInt64Add
- RtlPtrdiffTAdd
- RtlSSIZETAdd

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongLongMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **LONGLONG** by another.

Syntax

```
NTSTATUS RtlLongLongMult(
  LONGLONG llMultiplicand,
  LONGLONG llMultiplier,
  LONGLONG *pllResult
);
```

Parameters

`llMultiplicand`

The value to be multiplied by *llMultiplier*.

`llMultiplier`

The value by which to multiply *llMultiplicand*.

`pllResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrMult
- RtlLongPtrMult
- RtlLong64Mult
- RtlInt64Mult
- RtlPtrdiffTMult
- RtlSSIZETMult

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongLongSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **LONGLONG** from another.

Syntax

```
NTSTATUS RtlLongLongSub(
  LONGLONG llMinuend,
  LONGLONG llSubtrahend,
  LONGLONG *pllResult
);
```

Parameters

`llMinuend`

The value from which *llSubtrahend* is subtracted.

`llSubtrahend`

The value to subtract from *llMinuend*.

`pllResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrSub
- RtlLongPtrSub
- RtlLong64Sub
- RtlInt64Sub
- RtlPtrdiffTSub
- RtlSSIZETSub

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongLongToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlLongLongToChar(
  LONGLONG llOperand,
  CHAR     *pch
);
```

Parameters

`llOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLong64ToChar
- RtlInt64ToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **INT**.

Syntax

```
NTSTATUS RtlLongLongToInt(
  LONGLONG llOperand,
  INT      *piResult
);
```

Parameters

`llOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToInt
- RtlLongPtrToInt
- RtlLongLongToInt32
- RtlLongLongToIntPtr
- RtlLong64ToInt
- RtlLong64ToInt32
- RtlInt64ToInt
- RtlInt64ToInt32

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongLongToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlLongLongToInt8(
  LONGLONG llOperand,
  INT8     *pi8Result
);
```

Parameters

`llOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLong64ToInt8
- RtlInt64ToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlLongLongToIntPtr(
  LONGLONG llOperand,
  INT_PTR  *piResult
);
```

Parameters

`llOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToPtrdiffT
- RtlLong64ToIntPtr
- RtlLong64ToPtrdiffT
- RtlInt64ToIntPtr
- RtlInt64ToPtrdiffT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlLongLongToLong(
  LONGLONG llOperand,
  LONG     *plResult
);
```

Parameters

`llOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToLong
- RtlLongPtrToLong
- RtlLongLongToLongPtr
- RtlLong64ToLong
- RtlInt64ToLong

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlLongLongToLongPtr(
  LONGLONG llOperand,
  LONG_PTR *plResult
);
```

Parameters

`llOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToSSIZET
- RtlLong64ToLongPtr
- RtlLong64ToSSIZET
- RtlInt64ToLongPtr
- RtlInt64ToSSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlLongLongToShort(
  LONGLONG llOperand,
  SHORT    *psResult
);
```

Parameters

`llOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToInt16
- RtlLong64ToShort
- RtlLong64ToInt16
- RtlInt64ToShort
- RtlInt64ToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlLongLongToUChar(
  LONGLONG llOperand,
  UCHAR    *pch
);
```

Parameters

`llOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLong64ToUChar
- RtlInt64ToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlLongLongToUInt(
  LONGLONG llOperand,
  UINT     *puResult
);
```

Parameters

`llOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToUInt
- RtlLongPtrToUInt
- RtlLongLongToUInt32
- RtlLongLongToUIntPtr
- RtlLong64ToUInt
- RtlLong64ToUInt32
- RtlInt64ToUInt
- RtlInt64ToUInt32
- RtlLongLongToSizeT
- RtlLong64ToUIntPtr
- RtlLong64ToSizeT
- RtlInt64ToUIntPtr
- RtlInt64ToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **UNIT8**.

Syntax

```
NTSTATUS RtlLongLongToUInt8(
  LONGLONG llOperand,
  UINT8    *pu8Result
);
```

Parameters

`llOperand`

The value to be converted.

`pu8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToByte
- RtlLong64ToUInt8
- RtlLong64ToByte
- RtlInt64ToUInt8
- RtlInt64ToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlLongLongToULong(
  LONGLONG llOperand,
  ULONG    *pulResult
);
```

Parameters

`llOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToULong
- RtlLongPtrToULong
- RtlLongLongToULongPtr
- RtlLongLongToDWord
- RtlLong64ToULong
- RtlLong64ToDWord
- RtlInt64ToULong
- RtlInt64ToDWord

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongLongToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **LONGLONG**.

Syntax

```
NTSTATUS RtlLongLongToULongLong(
  LONGLONG  llOperand,
  ULONGLONG *pullResult
);
```

Parameters

`llOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlIntPtrToUIntPtr
- RtlIntPtrToULongPtr
- RtlIntPtrToULongLong
- RtlLongLongToUIntPtr
- RtlLongLongToULongPtr
- RtlLongLongToDWordLong
- RtlLongLongToULong64
- RtlLongLongToDWord64
- RtlLongLongToUInt64
- RtlLong64ToULongLong
- RtlInt64ToULongLong
- RtlInt64ToDWordLong
- RtlInt64ToULong64
- RtlInt64ToDWord64
- RtlInt64ToUInt64

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongLongToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONGLONG** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlLongLongToUShort(
  LONGLONG llOperand,
  USHORT   *pusResult
);
```

Parameters

`llOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToUInt16
- RtlLongLongToWord
- RtlLong64ToUShort
- RtlLong64ToUInt16
- RtlLong64ToWord
- RtlInt64ToUShort
- RtlInt64ToUInt16
- RtlInt64ToWord

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **LONG** by another.

Syntax

```
NTSTATUS RtlLongMult(
  LONG lMultiplicand,
  LONG lMultiplier,
  LONG *plResult
);
```

Parameters

`lMultiplicand`

The value to be multiplied by *lMultiplier*.

`lMultiplier`

The value by which to multiply *lMultiplicand*.

`plResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlLongPtrAdd(
  LONG_PTR lAugend,
  LONG_PTR lAddend,
  LONG_PTR *plResult
);
```

Parameters

`lAugend`

The first value in the equation.

`lAddend`

The value to add to *u8Augend*.

`plResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **LONG_PTR** by another.

Syntax

```
NTSTATUS RtlLongPtrMult(
  LONG_PTR lMultiplicand,
  LONG_PTR lMultiplier,
  LONG_PTR *plResult
);
```

Parameters

`lMultiplicand`

The value to be multiplied by *lMultiplier*.

`lMultiplier`

The value by which to multiply *lMultiplicand*.

`plResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **LONG_PTR** from another.

Syntax

```
NTSTATUS RtlLongPtrSub(
  LONG_PTR lMinuend,
  LONG_PTR lSubtrahend,
  LONG_PTR *plResult
);
```

Parameters

`lMinuend`

The value from which *lSubtrahend* is subtracted.

`lSubtrahend`

The value to subtract from *lMinuend*.

`plResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlLongPtrToChar(
  LONG_PTR lOperand,
  CHAR     *pch
);
```

Parameters

`lOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSSIZETToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **INT**.

Syntax

```
NTSTATUS RtlLongPtrToInt(
  LONG_PTR lOperand,
  INT      *piResult
);
```

Parameters

`lOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToInt32
- RtlSSIZETToInt
- RtlSSIZETToInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlLongPtrToInt8(
  LONG_PTR lOperand,
  INT8     *pi8Result
);
```

Parameters

`lOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSSIZETToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlLongPtrToIntPtr(
  LONG_PTR lOperand,
  INT_PTR  *piResult
);
```

Parameters

`lOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSSIZETToIntPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlLongPtrToLong(
  LONG_PTR lOperand,
  LONG     *plResult
);
```

Parameters

`lOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSSIZETToLong

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlLongPtrToShort(
  LONG_PTR lOperand,
  SHORT    *psResult
);
```

Parameters

`lOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToInt16
- RtlSSIZETToShort
- RtlSSIZETToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlLongPtrToUChar(
  LONG_PTR lOperand,
  UCHAR    *pch
);
```

Parameters

`lOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSSIZETToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlLongPtrToUInt(
  LONG_PTR lOperand,
  UINT     *puResult
);
```

Parameters

`lOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToUInt32
- RtlSSIZETToUInt
- RtlSSIZETToUInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlLongPtrToUInt8(
  LONG_PTR lOperand,
  UINT8    *pui8Result
);
```

Parameters

`lOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToByte
- RtlSSIZETToUInt8
- RtlSSIZETToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlLongPtrToUIntPtr(
  LONG_PTR lOperand,
  UINT_PTR *puResult
);
```

Parameters

`lOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToSizeT
- RtlSSIZETToUIntPtr
- RtlSSIZETToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlLongPtrToULong(
  LONG_PTR lOperand,
  ULONG    *pulResult
);
```

Parameters

`lOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToDWord
- RtlSSIZETToULong
- RtlSSIZETToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlLongPtrToULongLong(
  LONG_PTR  lOperand,
  ULONGLONG *pullResult
);
```

Parameters

`lOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToDWordLong
- RtlLongPtrToULong64
- RtlLongPtrToDWord64
- RtlLongPtrToUInt64
- RtlSSIZETToULongLong
- RtlSSIZETToDWordLong
- RtlSSIZETToULong64
- RtlSSIZETToDWord64
- RtlSSIZETToUInt64

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlLongPtrToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlLongPtrToULongPtr(
  LONG_PTR  lOperand,
  ULONG_PTR *pulResult
);
```

Parameters

`lOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToDWordPtr
- RtlLongPtrToSIZET
- RtlSSIZETToULongPtr
- RtlSSIZETToDWordPtr
- RtlSSIZETToSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongPtrToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG_PTR** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlLongPtrToUShort(
  LONG_PTR lOperand,
  USHORT   *pusResult
);
```

Parameters

`lOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongPtrToUInt16
- RtlLongPtrToWord
- RtlSSIZETToUShort
- RtlSSIZETToUInt16
- RtlSSIZETToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **LONG** from another.

Syntax

```
NTSTATUS RtlLongSub(
  LONG lMinuend,
  LONG lSubtrahend,
  LONG *plResult
);
```

Parameters

`lMinuend`

The value from which *lSubtrahend* is subtracted.

`lSubtrahend`

The value to subtract from *lMinuend*.

`plResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlLongToChar(
  LONG lOperand,
  CHAR *pch
);
```

Parameters

`lOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **INT**.

Syntax

```
NTSTATUS RtlLongToInt(
  LONG lOperand,
  INT  *piResult
);
```

Parameters

`lOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToInt32
- RtlLongToIntPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlLongToInt8(
  LONG lOperand,
  INT8 *pi8Result
);
```

Parameters

`lOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlLongToIntPtr(
  LONG    lOperand,
  INT_PTR *piResult
);
```

Parameters

`lOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToPtrdiffT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlLongToShort(
  LONG  lOperand,
  SHORT *psResult
);
```

Parameters

`lOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlLongToUChar(
  LONG  lOperand,
  UCHAR *pch
);
```

Parameters

`lOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlLongToUInt(
  LONG lOperand,
  UINT *puResult
);
```

Parameters

`lOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToUInt32
- RtlLongToUIntPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlLongToUInt8(
  LONG  lOperand,
  UINT8 *pui8Result
);
```

Parameters

`lOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlLongToUIntPtr(
  LONG     lOperand,
  UINT_PTR *puResult
);
```

Parameters

`lOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlLongToULong(
  LONG  lOperand,
  ULONG *pulResult
);
```

Parameters

`lOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToULongPtr
- RtlLongToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlLongToULongLong(
  LONG      lOperand,
  ULONGLONG *pullResult
);
```

Parameters

`lOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToDWordLong
- RtlLongToULong64
- RtlLongToDWord64
- RtlLongToUInt64

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlLongToULongPtr(
  LONG      lOperand,
  ULONG_PTR *pulResult
);
```

Parameters

`lOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToDWordPtr
- RtlLongToSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlLongToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **LONG** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlLongToUShort(
  LONG   lOperand,
  USHORT *pusResult
);
```

Parameters

`lOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongToUInt16
- RtlLongToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlPtrdiffTAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **PTRDIFF_T**.

Syntax

```
NTSTATUS RtlPtrdiffTAdd(
  ptrdiff_t Augend,
  ptrdiff_t Addend,
  ptrdiff_t *pResult
);
```

Parameters

`Augend`

The first value in the equation.

`Addend`

The value to add to *Augend*.

`pResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlPtrdiffTMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **PTRDIFF_T** by another.

Syntax

```
NTSTATUS RtlPtrdiffTMult(
  ptrdiff_t Multiplicand,
  ptrdiff_t Multiplier,
  ptrdiff_t *pResult
);
```

Parameters

`Multiplicand`

The value to be multiplied by *Multiplier*.

`Multiplier`

The value by which to multiply *Multiplicand*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlPtrdiffTSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **PTRDIFF_T** from another.

Syntax

```
NTSTATUS RtlPtrdiffTSub(
  ptrdiff_t Minuend,
  ptrdiff_t Subtrahend,
  ptrdiff_t *pResult
);
```

Parameters

`Minuend`

The value from which *Subtrahend* is subtracted.

`Subtrahend`

The value to subtract from *Minuend*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **SHORT**.

Syntax

```
NTSTATUS RtlShortAdd(
  SHORT sAugend,
  SHORT sAddend,
  SHORT *psResult
);
```

Parameters

`sAugend`

The first value in the equation.

`sAddend`

The value to add to *sAugend*.

`psResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16Add

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **SHORT** by another.

Syntax

```
NTSTATUS RtlShortMult(
  SHORT sMultiplicand,
  SHORT sMultiplier,
  SHORT *psResult
);
```

Parameters

`sMultiplicand`

The value to be multiplied by *sMultiplier*.

`sMultiplier`

The value by which to multiply *sMultiplicand*.

`psResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16Mult

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **SHORT** from another.

Syntax

```
NTSTATUS RtlShortSub(
  SHORT sMinuend,
  SHORT sSubtrahend,
  SHORT *psResult
);
```

Parameters

`sMinuend`

The value from which *sSubtrahend* is subtracted.

`sSubtrahend`

The value to subtract from *sMinuend*.

`psResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16Sub

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlShortToChar(
  SHORT sOperand,
  CHAR  *pch
);
```

Parameters

`sOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16ToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToDWordPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **DWORD_PTR**.

Syntax

```
NTSTATUS RtlShortToDWordPtr(
  SHORT     sOperand,
  DWORD_PTR *pdwResult
);
```

Parameters

`sOperand`

The value to be converted.

`pdwResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

-

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlShortToInt8(
  SHORT sOperand,
  INT8  *pi8Result
);
```

Parameters

`sOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16ToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlShortToUChar(
  SHORT sOperand,
  UCHAR *pch
);
```

Parameters

`sOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlInt16ToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlShortToUInt(
  SHORT sOperand,
  UINT  *puResult
);
```

Parameters

`sOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToUInt32
- RtlInt16ToUInt
- RtlInt16ToUInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlShortToUInt8(
  SHORT sOperand,
  UINT8 *pui8Result
);
```

Parameters

`sOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToByte
- RtlInt16ToUInt8
- RtlInt16ToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlShortToUIntPtr(
  SHORT    sOperand,
  UINT_PTR *puResult
);
```

Parameters

`sOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToSizeT
- RtlInt16ToUIntPtr
- RtlInt16ToSizeT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlShortToULong(
  SHORT sOperand,
  ULONG *pulResult
);
```

Parameters

`sOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToDWord
- RtlInt16ToULong
- RtlInt16ToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToULongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlShortToULongLong(
  SHORT     sOperand,
  ULONGLONG *pullResult
);
```

Parameters

`sOperand`

The value to be converted.

`pullResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToDWordLong
- RtlShortToULong64
- RtlShortToDWord64
- RtlShortToUInt64
- RtlInt16ToULongLong
- RtlInt16ToDWordLong
- RtlInt16ToULong64
- RtlInt16ToDWord64
- RtlInt16ToUInt64

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlShortToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlShortToULongPtr(
  SHORT     sOperand,
  ULONG_PTR *pulResult
);
```

Parameters

`sOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlLongLongToDWordPtr
- RtlLongLongToSIZET
- RtlLong64ToULongPtr
- RtlLong64ToDWordPtr
- RtlLong64ToSIZET
- RtlInt64ToULongPtr
- RtlInt64ToDWordPtr
- RtlInt64ToSIZET
- RtlShortToSIZET
- RtlInt16ToULongPtr
- RtlInt16ToDWordPtr
- RtlInt16ToSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlShortToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **SHORT** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlShortToUShort(
  SHORT  sOperand,
  USHORT *pusResult
);
```

Parameters

`sOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlShortToUInt16
- RtlShortToWord
- RtlInt16ToUShort
- RtlInt16ToUInt16
- RtlInt16ToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSizeTAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **SIZE_T**.

Syntax

```
NTSTATUS RtlSizeTAdd(
  size_t Augend,
  size_t Addend,
  size_t *pResult
);
```

Parameters

`Augend`

The first value in the equation.

`Addend`

The value to add to *Augend*.

`pResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSizeTMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **SIZE_T** by another.

Syntax

```
NTSTATUS RtlSizeTMult(
  size_t Multiplicand,
  size_t Multiplier,
  size_t *pResult
);
```

Parameters

`Multiplicand`

The value to be multiplied by *Multiplier*.

`Multiplier`

The value by which to multiply *Multiplicand*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSizeTSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **SIZE_T** from another.

Syntax

```
NTSTATUS RtlSizeTSub(
  size_t Minuend,
  size_t Subtrahend,
  size_t *pResult
);
```

Parameters

`Minuend`

The value from which *Subtrahend* is subtracted.

`Subtrahend`

The value to subtract from *Minuend*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSSIZETAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **SSIZE_T**.

Syntax

```
NTSTATUS RtlSSIZETAdd(
  SSIZE_T Augend,
  SSIZE_T Addend,
  SSIZE_T *pResult
);
```

Parameters

`Augend`

The first value in the equation.

`Addend`

The value to add to *Augend*.

`pResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSSIZETMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **SSIZE_T** by another.

Syntax

```
NTSTATUS RtlSSIZETMult(
  SSIZE_T Multiplicand,
  SSIZE_T Multiplier,
  SSIZE_T *pResult
);
```

Parameters

`Multiplicand`

The value to be multiplied by *Multiplier*.

`Multiplier`

The value by which to multiply *Multiplicand*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlSSIZETSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **SSIZE_T** from another.

Syntax

```
NTSTATUS RtlSSIZETSub(
  SSIZE_T Minuend,
  SSIZE_T Subtrahend,
  SSIZE_T *pResult
);
```

Parameters

`Minuend`

The value from which *Subtrahend* is subtracted.

`Subtrahend`

The value to subtract from *Minuend*.

`pResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUInt8Add function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **UINT8**.

Syntax

```
NTSTATUS RtlUInt8Add(
  UINT8 u8Augend,
  UINT8 u8Addend,
  UINT8 *pu8Result
);
```

Parameters

`u8Augend`

The first value in the equation.

`u8Addend`

The value to add to *u8Augend*.

`pu8Result`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUInt8Mult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **UINT8** by another.

Syntax

```
NTSTATUS RtlUInt8Mult(
  UINT8 u8Multiplicand,
  UINT8 u8Multiplier,
  UINT8 *pu8Result
);
```

Parameters

`u8Multiplicand`

The value to be multiplied by *u8Multiplier*.

`u8Multiplier`

The value by which to multiply *u8Multiplicand*.

`pu8Result`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUInt8Sub function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUInt8Sub** routine subtracts one value of type **UINT8** from another.

Syntax

```
NTSTATUS RtlUInt8Sub(
  UINT8 u8Minuend,
  UINT8 u8Subtrahend,
  UINT8 *pu8Result
);
```

Parameters

`u8Minuend`

The value from which *u8Subtrahend* is subtracted.

`u8Subtrahend`

The value to subtract from *u8Minuend*.

`pu8Result`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

RtlUInt8Sub returns STATUS_SUCCESS if the routine is successful. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STA TUS _IN TEG ER_ OVE RFL OW	An arithmetic overflow occurred.

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUInt8ToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT8** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlUInt8ToChar(
  UINT8 u8Operand,
  CHAR  *pch
);
```

Parameters

`u8Operand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUInt8ToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT8** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlUInt8ToInt8(
  UINT8 u8Operand,
  INT8  *pi8Result
);
```

Parameters

`u8Operand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **UINT**.

Syntax

```
NTSTATUS RtlUIntAdd(
  UINT uAugend,
  UINT uAddend,
  UINT *puResult
);
```

Parameters

`uAugend`

The first value in the equation.

`uAddend`

The value to add to *uAugend*.

`puResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUInt32Add

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **UINT** by another.

Syntax

```
NTSTATUS RtlUIntMult(
  UINT uMultiplicand,
  UINT uMultiplier,
  UINT *puResult
);
```

Parameters

`uMultiplicand`

The value to be multiplied by *uMultiplier*.

`uMultiplier`

The value by which to multiply *uMultiplicand*.

`puResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUInt32Mult

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlUIntPtrAdd(
  UINT_PTR uAugend,
  UINT_PTR uAddend,
  UINT_PTR *puResult
);
```

Parameters

`uAugend`

The first value in the equation.

`uAddend`

The value to add to *uAugend*.

`puResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlDWordAdd
- RtlUIntPtrAdd
- RtlULongPtrAdd
- RtlDWordPtrAdd
- RtlSIZETAdd
- RtlDWordLongAdd
- RtlULong64Add
- RtlDWord64Add
- RtlUInt64Add

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **UINT_PTR** by another.

Syntax

```
NTSTATUS RtlUIntPtrMult(
  UINT_PTR uMultiplicand,
  UINT_PTR uMultiplier,
  UINT_PTR *puResult
);
```

Parameters

`uMultiplicand`

The value to be multiplied by *uMultiplier*.

`uMultiplier`

The value by which to multiply *uMultiplicand*.

`puResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrMult
- RtlULongPtrMult
- RtlDWordPtrMult
- RtlSizeTMult
- RtlSIZETMult
- RtlDWordLongMult
- RtlULong64Mult
- RtlDWord64Mult
- RtlUInt64Mult

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **UINT_PTR** from another.

Syntax

```
NTSTATUS RtlUIntPtrSub(
  UINT_PTR uMinuend,
  UINT_PTR uSubtrahend,
  UINT_PTR *puResult
);
```

Parameters

`uMinuend`

The value from which *uSubtrahend* is subtracted.

`uSubtrahend`

The value to subtract from *uMinuend*.

`puResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrSub
- RtlULongPtrSub
- RtlDWordPtrSub
- RtlSIZETSub
- RtlDWordLongSub
- RtlULong64Sub
- RtlDWord64Sub
- RtlUInt64Sub

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlUIntPtrToChar(
  UINT_PTR uOperand,
  CHAR     *pch
);
```

Parameters

`uOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSizeTToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **INT**.

Syntax

```
NTSTATUS RtlUIntPtrToInt(
  UINT_PTR uOperand,
  INT      *piResult
);
```

Parameters

`uOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToInt32
- RtlSizeTToInt
- RtlSizeTToInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToInt16 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **INT16**.

Syntax

```
NTSTATUS RtlUIntPtrToInt16(
  UINT_PTR uOperand,
  INT16    *pi16Result
);
```

Parameters

`uOperand`

The value to be converted.

`pi16Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlUIntPtrToInt8(
  UINT_PTR uOperand,
  INT8     *pi8Result
);
```

Parameters

`uOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSizeTToInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlUIntPtrToIntPtr(
  UINT_PTR uOperand,
  INT_PTR  *piResult
);
```

Parameters

`uOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToPtrdiffT
- RtlSizeTToIntPtr
- RtlSizeTToPtrdiffT

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlUIntPtrToLong(
  UINT_PTR uOperand,
  LONG     *plResult
);
```

Parameters

`uOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSizeTToLong

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToLongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **LONGLONG**.

Syntax

```
NTSTATUS RtlUIntPtrToLongLong(
  UINT_PTR uOperand,
  LONGLONG *pllResult
);
```

Parameters

`uOperand`

The value to be converted.

`pllResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToLong64
- RtlUIntPtrToInt64
- RtlSizeTToLongLong
- RtlSizeTToLong64
- RtlSizeTToInt64

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlUIntPtrToLongPtr(
  UINT_PTR uOperand,
  LONG_PTR *plResult
);
```

Parameters

`uOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToSSIZET
- RtlSizeTToLongPtr
- RtlSizeTToSSIZET

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlUIntPtrToShort(
  UINT_PTR uOperand,
  SHORT    *psResult
);
```

Parameters

`uOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSizeTToShort
- RtlSizeTToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlUIntPtrToUChar(
  UINT_PTR uOperand,
  UCHAR    *pch
);
```

Parameters

`uOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlSizeTToUChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlUIntPtrToUInt(
  UINT_PTR uOperand,
  UINT     *puResult
);
```

Parameters

`uOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToUInt32
- RtlSizeTToUInt
- RtlSizeTToUInt32

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToUInt16 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **UINT16**.

Syntax

```
NTSTATUS RtlUIntPtrToUInt16(
  UINT_PTR uOperand,
  UINT16   *pu16Result
);
```

Parameters

`uOperand`

The value to be converted.

`pu16Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlUIntPtrToUInt8(
  UINT_PTR uOperand,
  UINT8    *pu8Result
);
```

Parameters

`uOperand`

The value to be converted.

`pu8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToByte
- RtlSizeTToUInt8
- RtlSizeTToByte

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlUIntPtrToULong(
  UINT_PTR uOperand,
  ULONG    *pulResult
);
```

Parameters

`uOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToDWord
- RtlSizeTToULong
- RtlSizeTToDWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntPtrToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT_PTR** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlUIntPtrToUShort(
  UINT_PTR uOperand,
  USHORT   *pusResult
);
```

Parameters

`uOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToWord
- RtlSizeTToUShort
- RtlSizeTToUInt16
- RtlSizeTToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **UINT** from another.

Syntax

```
NTSTATUS RtlUIntSub(
  UINT uMinuend,
  UINT uSubtrahend,
  UINT *puResult
);
```

Parameters

`uMinuend`

The value from which *u8Subtrahend* is subtracted.

`uSubtrahend`

The value to subtract from *u8Minuend*.

`puResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUInt32Sub

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlUIntToChar(
  UINT uOperand,
  CHAR *pch
);
```

Parameters

`uOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **INT**.

Syntax

```
NTSTATUS RtlUIntToInt(
  UINT uOperand,
  INT  *piResult
);
```

Parameters

`uOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlUIntToInt8(
  UINT uOperand,
  INT8 *pi8Result
);
```

Parameters

`uOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlUIntToIntPtr(
  UINT    uOperand,
  INT_PTR *piResult
);
```

Parameters

`uOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlUIntToLong(
  UINT uOperand,
  LONG *plResult
);
```

Parameters

`uOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntToLongPtr
- RtlUInt32ToLong

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlUIntToLongPtr(
  UINT     uOperand,
  LONG_PTR *plResult
);
```

Parameters

`uOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntToSSIZET
- RtlUInt32ToLongPtr

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlUIntToShort(
  UINT  uOperand,
  SHORT *psResult
);
```

Parameters

`uOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntToInt16
- RtlUInt32ToShort
- RtlUInt32ToInt16

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlUIntToUChar(
  UINT  uOperand,
  UCHAR *pch
);
```

Parameters

`uOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUInt32ToChar

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlUIntToUInt8(
  UINT  uOperand,
  UINT8 *pui8Result
);
```

Parameters

`uOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntToByte
- RtlUInt32ToUInt8

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUIntToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **UINT** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlUIntToUShort(
  UINT   uOperand,
  USHORT *pusResult
);
```

Parameters

`uOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntToUInt16
- RtlUIntToWord
- RtlUInt32ToUShort
- RtlUInt32ToUInt16
- RtlUInt32ToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **ULONG**.

Syntax

```
NTSTATUS RtlULongAdd(
  ULONG ulAugend,
  ULONG ulAddend,
  ULONG *pulResult
);
```

Parameters

`ulAugend`

The first value in the equation.

`ulAddend`

The value to add to *ulAugend*.

`pulResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **ULONGLONG**.

Syntax

```
NTSTATUS RtlULongLongAdd(
  ULONGLONG ullAugend,
  ULONGLONG ullAddend,
  ULONGLONG *pullResult
);
```

Parameters

`ullAugend`

The first value in the equation.

`ullAddend`

The value to add to *ullAugend*.

`pullResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **ULONGLONG** by another.

Syntax

```
NTSTATUS RtlULongLongMult(
  ULONGLONG ullMultiplicand,
  ULONGLONG ullMultiplier,
  ULONGLONG *pullResult
);
```

Parameters

`ullMultiplicand`

The value to be multiplied by *ullMultiplier*.

`ullMultiplier`

The value by which to multiply *ullMultiplicand*.

`pullResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **ULONGLONG** from another.

Syntax

```
NTSTATUS RtlULongLongSub(
  ULONGLONG ullMinuend,
  ULONGLONG ullSubtrahend,
  ULONGLONG *pullResult
);
```

Parameters

`ullMinuend`

The value from which *ullSubtrahend* is subtracted.

`ullSubtrahend`

The value to subtract from *ullMinuend*.

`pullResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlULongLongToChar(
  ULONGLONG ullOperand,
  CHAR      *pch
);
```

Parameters

`ullOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **INT**.

Syntax

```
NTSTATUS RtlULongLongToInt(
  ULONGLONG ullOperand,
  INT       *piResult
);
```

Parameters

`ullOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlULongLongToInt8(
  ULONGLONG ullOperand,
  INT8      *pi8Result
);
```

Parameters

`ullOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlULongLongToLong(
  ULONGLONG ullOperand,
  LONG      *plResult
);
```

Parameters

`ullOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToLongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **LONGLONG**.

Syntax

```
NTSTATUS RtlULongLongToLongLong(
  ULONGLONG ullOperand,
  LONGLONG  *pllResult
);
```

Parameters

`ullOperand`

The value to be converted.

`pllResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUIntPtrToLongLong
- RtlULongPtrToLongLong
- RtlULongLongToIntPtr
- RtlULongLongToInt64
- RtlULongLongToLong64
- RtlDWordLongToLongLong
- RtlDWordLongToLong64
- RtlDWordLongToInt64
- RtlULong64ToLongLong
- RtlULong64ToLong64
- RtlULong64ToInt64
- RtlDWord64ToLongLong
- RtlDWord64ToLong64
- RtlDWord64ToInt64
- RtlUInt64ToLongLong

- RtlUInt64ToLong64
- RtlUInt64ToInt64

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlULongLongToLongPtr(
  ULONGLONG ullOperand,
  LONG_PTR  *plResult
);
```

Parameters

`ullOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlULongLongToSSIZET
- RtlDWordLongToLongPtr
- RtlDWordLongToSSIZET
- RtlULong64ToLongPtr
- RtlULong64ToSSIZET
- RtlDWord64ToLongPtr
- RtlDWord64ToSSIZET
- RtlUInt64ToLongPtr
- RtlUInt64ToSSIZET

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlULongLongToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlULongLongToShort(
  ULONGLONG ullOperand,
  SHORT     *psResult
);
```

Parameters

`ullOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlULongLongToUChar(
  ULONGLONG ullOperand,
  UCHAR     *pch
);
```

Parameters

`ullOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlULongLongToUInt(
  ULONGLONG ullOperand,
  UINT      *puResult
);
```

Parameters

`ullOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlULongLongToUInt8(
  ULONGLONG ullOperand,
  UINT8     *pu8Result
);
```

Parameters

`ullOperand`

The value to be converted.

`pu8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlULongLongToUIntPtr(
  ULONGLONG ullOperand,
  UINT_PTR  *puResult
);
```

Parameters

`ullOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlULongLongToULong(
  ULONGLONG ullOperand,
  ULONG     *pulResult
);
```

Parameters

`ullOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToULongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlULongLongToULongPtr(
  ULONGLONG ullOperand,
  ULONG_PTR *pulResult
);
```

Parameters

`ullOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongLongToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONGLONG** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlULongLongToUShort(
  ULONGLONG ullOperand,
  USHORT    *pusResult
);
```

Parameters

`ullOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlUInt64ToDWord
- RtlULongLongToUInt16
- RtlULongLongToWord
- RtlDWordLongToUShort
- RtlDWordLongToUInt16
- RtlDWordLongToWord
- RtlULong64ToUShort
- RtlULong64ToUInt16
- RtlULong64ToWord
- RtlDWord64ToUShort
- RtlDWord64ToUInt16
- RtlDWord64ToWord
- RtlUInt64ToUShort
- RtlUInt64ToUInt16
- RtlUInt64ToWord

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **ULONG** by another.

Syntax

```
NTSTATUS RtlULongMult(
  ULONG ulMultiplicand,
  ULONG ulMultiplier,
  ULONG *pulResult
);
```

Parameters

`ulMultiplicand`

The value to be multiplied by *ulMultiplier*.

`ulMultiplier`

The value by which to multiply *ulMultiplicand*.

`pulResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **ULONG_PTR**.

Syntax

```
NTSTATUS RtlULongPtrAdd(
  ULONG_PTR ulAugend,
  ULONG_PTR ulAddend,
  ULONG_PTR *pulResult
);
```

Parameters

`ulAugend`

The first value in the equation.

`ulAddend`

The value to add to *ulAugend*.

`pulResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **ULONG_PTR** by another.

Syntax

```
NTSTATUS RtlULongPtrMult(
  ULONG_PTR ulMultiplicand,
  ULONG_PTR ulMultiplier,
  ULONG_PTR *pulResult
);
```

Parameters

`ulMultiplicand`

The value to be multiplied by *Multiplier*.

`ulMultiplier`

The value by which to multiply *Multiplicand*.

`pulResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **ULONG_PTR** from another.

Syntax

```
NTSTATUS RtlULongPtrSub(
  ULONG_PTR ulMinuend,
  ULONG_PTR ulSubtrahend,
  ULONG_PTR *pulResult
);
```

Parameters

`ulMinuend`

The value from which *ulSubtrahend* is subtracted.

`ulSubtrahend`

The value to subtract from *ulMinuend*.

`pulResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlULongPtrToChar(
  ULONG_PTR ulOperand,
  CHAR      *pch
);
```

Parameters

`ulOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **INT**.

Syntax

```
NTSTATUS RtlULongPtrToInt(
  ULONG_PTR ulOperand,
  INT       *piResult
);
```

Parameters

`ulOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlULongPtrToInt8(
  ULONG_PTR ulOperand,
  INT8      *pi8Result
);
```

Parameters

`ulOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlULongPtrToIntPtr(
  ULONG_PTR ulOperand,
  INT_PTR   *piResult
);
```

Parameters

`ulOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlULongPtrToLong(
  ULONG_PTR ulOperand,
  LONG      *plResult
);
```

Parameters

`ulOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToLongLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **LONGLONG**.

Syntax

```
NTSTATUS RtlULongPtrToLongLong(
  ULONG_PTR ulOperand,
  LONGLONG  *pllResult
);
```

Parameters

`ulOperand`

The value to be converted.

`pllResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlULongPtrToLong64
- RtlULongPtrToInt64
- RtlDWordPtrToLongLong
- RtlDWordPtrToLong64
- RtlDWordPtrToInt64
- RtlSIZETToLongLong
- RtlSIZETToLong64
- RtlSIZETToInt64

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlULongPtrToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlULongPtrToLongPtr(
  ULONG_PTR ulOperand,
  LONG_PTR  *plResult
);
```

Parameters

`ulOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlULongPtrToShort(
  ULONG_PTR ulOperand,
  SHORT     *psResult
);
```

Parameters

`ulOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

This function uses the following alternate name:

- RtlULongPtrToUInt16
- RtlULongPtrToWord
- RtlDWordPtrToUShort
- RtlDWordPtrToUInt16
- RtlDWordPtrToWord
- RtlSIZETToUShort
- RtlSIZETToUInt16
- RtlSIZETToWord

Requirements

Target Platform	Desktop

Header	ntintsafe.h

RtlULongPtrToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlULongPtrToUChar(
  ULONG_PTR ulOperand,
  UCHAR     *pch
);
```

Parameters

`ulOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlULongPtrToUInt(
  ULONG_PTR ulOperand,
  UINT      *puResult
);
```

Parameters

`ulOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlULongPtrToUInt8(
  ULONG_PTR ulOperand,
  UINT8     *pui8Result
);
```

Parameters

`ulOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlULongPtrToUIntPtr(
  ULONG_PTR ulOperand,
  UINT_PTR  *puResult
);
```

Parameters

`ulOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToULong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **ULONG**.

Syntax

```
NTSTATUS RtlULongPtrToULong(
  ULONG_PTR ulOperand,
  ULONG     *pulResult
);
```

Parameters

`ulOperand`

The value to be converted.

`pulResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongPtrToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlULongPtrToUShort(
  ULONG_PTR ulOperand,
  USHORT    *pusResult
);
```

Parameters

`ulOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **ULONG** from another.

Syntax

```
NTSTATUS RtlULongSub(
  ULONG ulMinuend,
  ULONG ulSubtrahend,
  ULONG *pulResult
);
```

Parameters

`ulMinuend`

The value from which *ulSubtrahend* is subtracted.

`ulSubtrahend`

The value to subtract from *ulMinuend*.

`pulResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlULongToChar(
  ULONG ulOperand,
  CHAR  *pch
);
```

Parameters

`ulOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **INT**.

Syntax

```
NTSTATUS RtlULongToInt(
  ULONG ulOperand,
  INT   *piResult
);
```

Parameters

`ulOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlULongToInt8(
  ULONG ulOperand,
  INT8  *pi8Result
);
```

Parameters

`ulOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **INT_PTR**.

Syntax

```
NTSTATUS RtlULongToIntPtr(
  ULONG   ulOperand,
  INT_PTR *piResult
);
```

Parameters

`ulOperand`

The value to be converted.

`piResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToLong function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **LONG**.

Syntax

```
NTSTATUS RtlULongToLong(
  ULONG ulOperand,
  LONG  *plResult
);
```

Parameters

`ulOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToLongPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **LONG_PTR**.

Syntax

```
NTSTATUS RtlULongToLongPtr(
  ULONG    ulOperand,
  LONG_PTR *plResult
);
```

Parameters

`ulOperand`

The value to be converted.

`plResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlULongToShort(
  ULONG ulOperand,
  SHORT *psResult
);
```

Parameters

`ulOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlULongToUChar(
  ULONG ulOperand,
  UCHAR *pch
);
```

Parameters

`ulOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToUInt function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **UINT**.

Syntax

```
NTSTATUS RtlULongToUInt(
  ULONG ulOperand,
  UINT  *puResult
);
```

Parameters

`ulOperand`

The value to be converted.

`puResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlULongToUInt8(
  ULONG ulOperand,
  UINT8 *pui8Result
);
```

Parameters

`ulOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToUIntPtr function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG_PTR** to a value of type **UINT_PTR**.

Syntax

```
NTSTATUS RtlULongToUIntPtr(
  ULONG    ulOperand,
  UINT_PTR *puiResult
);
```

Parameters

`ulOperand`

The value to be converted.

`puiResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlULongToUShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **ULONG** to a value of type **USHORT**.

Syntax

```
NTSTATUS RtlULongToUShort(
  ULONG  ulOperand,
  USHORT *pusResult
);
```

Parameters

`ulOperand`

The value to be converted.

`pusResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortAdd function

1/25/2019 • 2 minutes to read • Edit Online

Adds two values of type **USHORT**.

Syntax

```
NTSTATUS RtlUShortAdd(
  USHORT usAugend,
  USHORT usAddend,
  USHORT *pusResult
);
```

Parameters

`usAugend`

The first value in the equation.

`usAddend`

The value to add to *usAugend*.

`pusResult`

A pointer to the sum. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortMult function

1/25/2019 • 2 minutes to read • Edit Online

Multiplies one value of type **USHORT** by another.

Syntax

```
NTSTATUS RtlUShortMult(
  USHORT usMultiplicand,
  USHORT usMultiplier,
  USHORT *pusResult
);
```

Parameters

`usMultiplicand`

The value to be multiplied by *usMultiplier*.

`usMultiplier`

The value by which to multiply *usMultiplicand*.

`pusResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortSub function

1/25/2019 • 2 minutes to read • Edit Online

Subtracts one value of type **USHORT** from another.

Syntax

```
NTSTATUS RtlUShortSub(
  USHORT usMinuend,
  USHORT usSubtrahend,
  USHORT *pusResult
);
```

Parameters

`usMinuend`

The value from which *usSubtrahend* is subtracted.

`usSubtrahend`

The value to subtract from *usMinuend*.

`pusResult`

A pointer to the result. If the operation results in a value that overflows or underflows the capacity of the type, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide arithmetic operations and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortToChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **USHORT** to a value of type **CHAR**.

Syntax

```
NTSTATUS RtlUShortToChar(
  USHORT usOperand,
  CHAR   *pch
);
```

Parameters

`usOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortToInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **USHORT** to a value of type **INT8**.

Syntax

```
NTSTATUS RtlUShortToInt8(
  USHORT usOperand,
  INT8   *pi8Result
);
```

Parameters

`usOperand`

The value to be converted.

`pi8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortToShort function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **USHORT** to a value of type **SHORT**.

Syntax

```
NTSTATUS RtlUShortToShort(
  USHORT usOperand,
  SHORT  *psResult
);
```

Parameters

`usOperand`

The value to be converted.

`psResult`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortToUChar function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **USHORT** to a value of type **UCHAR**.

Syntax

```
NTSTATUS RtlUShortToUChar(
  USHORT usOperand,
  UCHAR  *pch
);
```

Parameters

`usOperand`

The value to be converted.

`pch`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

RtlUShortToUInt8 function

1/25/2019 • 2 minutes to read • Edit Online

Converts a value of type **USHORT** to a value of type **UINT8**.

Syntax

```
NTSTATUS RtlUShortToUInt8(
  USHORT usOperand,
  UINT8  *pui8Result
);
```

Parameters

`usOperand`

The value to be converted.

`pui8Result`

A pointer to the converted value. In the case where the conversion causes a truncation of the original value, the function returns STATUS_INTEGER_OVERFLOW and this parameter is not valid.

Return Value

None

Remarks

This is one of a set of inline functions designed to provide type conversions and perform validity checks with minimal impact on performance.

Requirements

Target Platform	Desktop
Header	ntintsafe.h

ntpoapi.h header

1/8/2019 • 2 minutes to read • Edit Online

This header contains functions and data structures that an application can use power management. For more information, see:

- About Power Management ntpoapi.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
NtPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.

Structures

TITLE	DESCRIPTION
CM_POWER_DATA	The CM_POWER_DATA structure contains information about a device's power management state and capabilities.
COUNTED_REASON_CONTEXT	The COUNTED_REASON_CONTEXT structure contains one or more strings that give reasons for a power request.
PO_SPR_ACTIVE_SESSION_DATA	Allow drivers to register/unregister for SPR Active session start and end notifications.
POWER_PLATFORM_INFORMATION	The POWER_PLATFORM_INFORMATION structure contains information about the power capabilities of the system.
POWER_STATE	The POWER_STATE union specifies a system power state value or a device power state value.
SYSTEM_POWER_STATE_CONTEXT	The SYSTEM_POWER_STATE_CONTEXT structure is a partially opaque system structure that contains information about the previous system power states of a computer.

Enumerations

TITLE	DESCRIPTION
DEVICE_POWER_STATE	The DEVICE_POWER_STATE enumeration type indicates a device power state.
MONITOR_DISPLAY_STATE	Indicates the power state of the monitor being displayed on.
POWER_ACTION	The POWER_ACTION enumeration identifies the system power actions that can occur on a computer.

TITLE	DESCRIPTION
POWER_INFORMATION_LEVEL	Indicates power level information.
POWER_MONITOR_REQUEST_REASON	Defines values for power transitions reasons for a monitor.
POWER_REQUEST_TYPE	The POWER_REQUEST_TYPE enumeration indicates the power request type.
POWER_STATE_TYPE	The POWER_STATE_TYPE enumeration type indicates that a power state value is a system power state or a device power state.
SYSTEM_POWER_STATE	The SYSTEM_POWER_STATE enumeration type is used to indicate a system power state.

1/30/2019 • 2 minutes to read • Edit Online

UID: NE:ntpoapi.__unnamed_enum_0 title: *PPOWER_ACTION (ntpoapi.h) description: The POWER_ACTION enumeration identifies the system power actions that can occur on a computer. old-location: wdf\power_action.htm tech.root: kernel ms.assetid: 0c4a5eb8-d364-4e5d-9d2f-2605c8c34f63 ms.date: 02/26/2018 ms.keywords:* "PPOWER_ACTION, POWER_ACTION, POWER_ACTION enumeration, PPOWER_ACTION, PPOWER_ACTION enumeration pointer, PowerActionHibernate, PowerActionNone, PowerActionReserved, PowerActionShutdown, PowerActionShutdownOff, PowerActionShutdownReset, PowerActionSleep, PowerActionWarmEject, umdf.power_action, umdfstructs_02db2a83-1907-4aa2-9cdd-af3477e78ddd.xml, wdf.power_action, wudfddi/POWER_ACTION, wudfddi/PPOWER_ACTION, wudfddi/PowerActionHibernate, wudfddi/PowerActionNone, wudfddi/PowerActionReserved, wudfddi/PowerActionShutdown, wudfddi/PowerActionShutdownOff, wudfddi/PowerActionShutdownReset, wudfddi/PowerActionSleep, wudfddi/PowerActionWarmEject" ms.topic: enum req.header: ntpoapi.h req.include-header: Wudfddi.h, Ntpoapi.h, Wdm.h req.target-type: Windows req.target-min-winverclnt: req.target-min-winversvr: req.kmdf-ver: req.umdf-ver: 1.9 req.ddi-compliance: req.unicode-ansi: req.idl: req.max-support: Unavailable in UMDF 2.0 and later. req.namespace: req.assembly: req.type-library: req.lib: req.dll: req.irql: topic_type:

- APIRef
- kbSyntax api_type:
- HeaderDef api_location:
- Wudfddi.h api_name:
- POWER_ACTION product:
- Windows targetos: Windows

req.typenames: POWER_ACTION, *PPOWER_ACTION

POWER_ACTION Enumeration

[**Warning:** UMDF 2 is the latest version of UMDF and supersedes UMDF 1. All new UMDF drivers should be written using UMDF 2. No new features are being added to UMDF 1 and there is limited support for UMDF 1 on newer versions of Windows 10. Universal Windows drivers must use UMDF 2. For more info, see Getting Started with UMDF.]

The **POWER_ACTION** enumeration identifies the system power actions that can occur on a computer.

Syntax

```
typedef enum  {
  PowerActionNone           = 0,
  PowerActionReserved       = ( PowerActionNone + 1 ),
  PowerActionSleep          = ( PowerActionReserved + 1 ),
  PowerActionHibernate      = ( PowerActionSleep + 1 ),
  PowerActionShutdown       = ( PowerActionHibernate + 1 ),
  PowerActionShutdownReset  = ( PowerActionShutdown + 1 ),
  PowerActionShutdownOff    = ( PowerActionShutdownReset + 1 ),
  PowerActionWarmEject      = ( PowerActionShutdownOff + 1 )
} POWER_ACTION, *PPOWER_ACTION;
```

Constants

PowerActionNone	No power action is taking place.
PowerActionReserved	Reserved for system use.
PowerActionSleep	The computer is entering a system sleeping (S1, S2, or S3) state.
PowerActionHibernate	The computer is entering its hibernation (S4) state.
PowerActionShutdown	The computer is entering its shutdown (S5) state. After all devices have entered their off (D3) state, the computer remains powered on until an administrator presses the power button.
PowerActionShutdownReset	The computer is entering its shutdown (S5) state. After all devices have entered their off (D3) state, the computer automatically powers off and then immediately restarts and returns to its working (S0) state.
PowerActionShutdownOff	The computer is entering its shutdown (S5) state. After all devices have entered their off (D3) state, the computer automatically powers off.
PowerActionWarmEject	The computer is being ejected from an ACPI-compatible dock device. Typically, the computer's power state does not change.
PowerActionDisplayOff	

Remarks

The **POWER_ACTION** enumeration is used as the return value for IWDFDevice2::GetSystemPowerAction.

Requirements

Minimum UMDF version	1.9
Header	ntpoapi.h (include Wudfddi.h, Ntpoapi.h, Wdm.h)

See Also

IWDFDevice2::GetSystemPowerAction

POWER_INFORMATION_LEVEL Enumeration

1/25/2019 • 3 minutes to read • Edit Online

Indicates power level information.

Syntax

```
typedef enum  {
  SystemPowerPolicyAc,
  SystemPowerPolicyDc,
  VerifySystemPolicyAc,
  VerifySystemPolicyDc,
  SystemPowerCapabilities,
  SystemBatteryState,
  SystemPowerStateHandler,
  ProcessorStateHandler,
  SystemPowerPolicyCurrent,
  AdministratorPowerPolicy,
  SystemReserveHiberFile,
  ProcessorInformation,
  SystemPowerInformation,
  ProcessorStateHandler2,
  LastWakeTime,
  LastSleepTime,
  SystemExecutionState,
  SystemPowerStateNotifyHandler,
  ProcessorPowerPolicyAc,
  ProcessorPowerPolicyDc,
  VerifyProcessorPowerPolicyAc,
  VerifyProcessorPowerPolicyDc,
  ProcessorPowerPolicyCurrent,
  SystemPowerStateLogging,
  SystemPowerLoggingEntry,
  SetPowerSettingValue,
  NotifyUserPowerSetting,
  PowerInformationLevelUnused0,
  SystemMonitorHiberBootPowerOff,
  SystemVideoState,
  TraceApplicationPowerMessage,
  TraceApplicationPowerMessageEnd,
  ProcessorPerfStates,
  ProcessorIdleStates,
  ProcessorCap,
  SystemWakeSource,
  SystemHiberFileInformation,
  TraceServicePowerMessage,
  ProcessorLoad,
  PowerShutdownNotification,
  MonitorCapabilities,
  SessionPowerInit,
  SessionDisplayState,
  PowerRequestCreate,
  PowerRequestAction,
  GetPowerRequestList,
  ProcessorInformationEx,
  NotifyUserModeLegacyPowerEvent,
  GroupPark,
  ProcessorIdleDomains,
  WakeTimerList,
  SystemHiberFileSize,
  ProcessorIdleStatesHv,
```

```
    ProcessorPerfStatesHv,
    ProcessorPerfCapHv,
    ProcessorSetIdle,
    LogicalProcessorIdling,
    UserPresence,
    PowerSettingNotificationName,
    GetPowerSettingValue,
    IdleResiliency,
    SessionRITState,
    SessionConnectNotification,
    SessionPowerCleanup,
    SessionLockState,
    SystemHiberbootState,
    PlatformInformation,
    PdcInvocation,
    MonitorInvocation,
    FirmwareTableInformationRegistered,
    SetShutdownSelectedTime,
    SuspendResumeInvocation,
    PlmPowerRequestCreate,
    ScreenOff,
    CsDeviceNotification,
    PlatformRole,
    LastResumePerformance,
    DisplayBurst,
    ExitLatencySamplingPercentage,
    RegisterSpmPowerSettings,
    PlatformIdleStates,
    ProcessorIdleVeto,
    PlatformIdleVeto,
    SystemBatteryStatePrecise,
    ThermalEvent,
    PowerRequestActionInternal,
    BatteryDeviceState,
    PowerInformationInternal,
    ThermalStandby,
    SystemHiberFileType,
    PhysicalPowerButtonPress,
    QueryPotentialDripsConstraint,
    EnergyTrackerCreate,
    EnergyTrackerQuery,
    UpdateBlackBoxRecorder,
    PowerInformationLevelMaximum
} POWER_INFORMATION_LEVEL;
```

Constants

SystemPowerPolicyAc	Indicates SystemPowerPolicyAc.
SystemPowerPolicyDc	Indicates SystemPowerPolicyDc.
VerifySystemPolicyAc	Indicates VerifySystemPolicyAc.
VerifySystemPolicyDc	Indicates VerifySystemPolicyDc.
SystemPowerCapabilities	Indicates the power capabilities of the system.
SystemBatteryState	Indicates the system's battery state.
SystemPowerStateHandler	Indicates the system's power state handler.

ProcessorStateHandler	Indicates the processor state handler.
SystemPowerPolicyCurrent	Indicates the system's current power policy.
AdministratorPowerPolicy	Indicates the administrator's power policy.
SystemReserveHiberFile	Indicates the SystemReserveHiberFile.
ProcessorInformation	Indicates the processor information.
SystemPowerInformation	Indicates the system power information.
ProcessorStateHandler2	Indicates the processor state handler.
LastWakeTime	Indicates the last wake time.
LastSleepTime	Indicates the last sleep time.
SystemExecutionState	Indicates the system execution state.
SystemPowerStateNotifyHandler	Indicates the system power state notify handler.
ProcessorPowerPolicyAc	Indicates ProcessorPowerPolicyAc.
ProcessorPowerPolicyDc	Indicates ProcessorPowerPolicyDc.
VerifyProcessorPowerPolicyAc	Indicates VerifyProcessorPowerPolicyAc.
VerifyProcessorPowerPolicyDc	Indicates VerifyProcessorPowerPolicyDc.
ProcessorPowerPolicyCurrent	Indicates the current processor power policy.
SystemPowerStateLogging	Indicates SystemPowerStateLogging.
SystemPowerLoggingEntry	Indicates SystemPowerLoggingEntry.
SetPowerSettingValue	Indicates that the power setting value is set.
NotifyUserPowerSetting	Indicates that the user should be notified of the power setting.
PowerInformationLevelUnused0	Indicates that the power information level is unused.
SystemMonitorHiberBootPowerOff	Indicates that the system monitor boot power is off.
SystemVideoState	Indicates the system video state.
TraceApplicationPowerMessage	Indicates the trace application power message.
TraceApplicationPowerMessageEnd	Indicates the end of the trace application power message.
ProcessorPerfStates	Indicates the processor performance states.

ProcessorIdleStates	Indicates the processor idle states.
ProcessorCap	Indicates the processor cap.
SystemWakeSource	Indicates the system wake source.
SystemHiberFileInformation	Indicates the system's hibernation file information.
TraceServicePowerMessage	Indicates the trace service power message.
ProcessorLoad	Indicates the processor load.
PowerShutdownNotification	Indicates the power shutdown notification.
MonitorCapabilities	Indicates the monitor's capabilities.
SessionPowerInit	Indicates the session power has been initialized.
SessionDisplayState	Indicates the session display state.
PowerRequestCreate	Indicates that a power request has been created.
PowerRequestAction	Indicates the action of the power request.
GetPowerRequestList	Indicates that the power request list should be queued.
ProcessorInformationEx	Indicates ProcessorInformationEx.
NotifyUserModeLegacyPowerEvent	Indicates that a notification should be created for the user mode legacy power event.
GroupPark	Indicates the group park.
ProcessorIdleDomains	Indicates the processor's idle domains.
WakeTimerList	Indicates the wake timer list.
SystemHiberFileSize	Indicates the system's hibernation file size.
ProcessorIdleStatesHv	Indicates the processor's idle states.
ProcessorPerfStatesHv	Indicates the processor's performance states.
ProcessorPerfCapHv	Indicates the processor's performance capabilities.
ProcessorSetIdle	Indicates that the processor has been set to idle.
LogicalProcessorIdling	Indicates that the processor is idling.
UserPresence	Indicates the user presence.
PowerSettingNotificationName	Indicates the power setting notification name.

GetPowerSettingValue	Indicates that the power setting value should be queued.
IdleResiliency	Indicates the idle resiliency.
SessionRITState	Indicates the session's RIT state.
SessionConnectNotification	Indicates the session's connect notification.
SessionPowerCleanup	Indicates the session's power cleanup.
SessionLockState	Indicates the session's lock state.
SystemHiberbootState	Indicates the system's hibernation boot state.
PlatformInformation	Indicates the platform information.
PdcInvocation	Indicates the pdc invocation.
MonitorInvocation	Indicates the monitor invocation.
FirmwareTableInformationRegistered	Indicates the registered firmware table information.
SetShutdownSelectedTime	Indicates that the shutdown time should be set.
SuspendResumeInvocation	Indicates SuspendResumeInvocation.
PlmPowerRequestCreate	Indicates that the power request has been created.
ScreenOff	Indicates that the screen is off.
CsDeviceNotification	Indicates the device notification.
PlatformRole	Indicates the platform role.
LastResumePerformance	Indicates the last time performance was resumed.
DisplayBurst	Indicates display burst.
ExitLatencySamplingPercentage	Indicates the latency sampling percentage.
RegisterSpmPowerSettings	Indicates that the power settings are registered.
PlatformIdleStates	Indicates the platform's idle states.
ProcessorIdleVeto	Indicates the processor's idle veto.
PlatformIdleVeto	Indicates the platform's idle veto.
SystemBatteryStatePrecise	Indicates the system's battery state.
ThermalEvent	Indicates the thermal event.

PowerRequestActionInternal	Indicates the internal power request action.
BatteryDeviceState	Indicates the battery's device state.
PowerInformationInternal	Indicates the internal power information.
ThermalStandby	Indicates thermal standby.
SystemHiberFileType	Indicates the system's hibernation file type.
PhysicalPowerButtonPress	Indicates a physical power button press.
QueryPotentialDripsConstraint	Indicates the potential drips constraint.
EnergyTrackerCreate	Indicates that the energy tracker is created.
EnergyTrackerQuery	Indicates that the energy tracker is queried.
UpdateBlackBoxRecorder	Indicates that the black box recorder is updated.
PowerInformationLevelMaximum	Indicates the maximum power level.

Requirements

Header	ntpoapi.h (include Ntpoapi.h)

POWER_MONITOR_REQUEST_REASON Enumeration

1/25/2019 • 2 minutes to read • Edit Online

Defines values for power transitions reasons for a monitor.

Syntax

```
typedef enum  {
  MonitorRequestReasonUnknown,
  MonitorRequestReasonPowerButton,
  MonitorRequestReasonRemoteConnection,
  MonitorRequestReasonScMonitorpower,
  MonitorRequestReasonUserInput,
  MonitorRequestReasonAcDcDisplayBurst,
  MonitorRequestReasonUserDisplayBurst,
  MonitorRequestReasonPoSetSystemState,
  MonitorRequestReasonSetThreadExecutionState,
  MonitorRequestReasonFullWake,
  MonitorRequestReasonSessionUnlock,
  MonitorRequestReasonScreenOffRequest,
  MonitorRequestReasonIdleTimeout,
  MonitorRequestReasonPolicyChange,
  MonitorRequestReasonSleepButton,
  MonitorRequestReasonLid,
  MonitorRequestReasonBatteryCountChange,
  MonitorRequestReasonGracePeriod,
  MonitorRequestReasonPnP,
  MonitorRequestReasonDP,
  MonitorRequestReasonSxTransition,
  MonitorRequestReasonSystemIdle,
  MonitorRequestReasonNearProximity,
  MonitorRequestReasonThermalStandby,
  MonitorRequestReasonResumePdc,
  MonitorRequestReasonResumeS4,
  MonitorRequestReasonTerminal,
  MonitorRequestReasonPdcSignal,
  MonitorRequestReasonAcDcDisplayBurstSuppressed,
  MonitorRequestReasonSystemStateEntered,
  MonitorRequestReasonWinrt,
  MonitorRequestReasonUserInputKeyboard,
  MonitorRequestReasonUserInputMouse,
  MonitorRequestReasonUserInputTouch,
  MonitorRequestReasonUserInputPen,
  MonitorRequestReasonUserInputAccelerometer,
  MonitorRequestReasonUserInputHid,
  MonitorRequestReasonUserInputPoUserPresent,
  MonitorRequestReasonUserInputSessionSwitch,
  MonitorRequestReasonUserInputInitialization,
  MonitorRequestReasonPdcSignalWindowsMobilePwrNotif,
  MonitorRequestReasonPdcSignalWindowsMobileShell,
  MonitorRequestReasonPdcSignalHeyCortana,
  MonitorRequestReasonPdcSignalHolographicShell,
  MonitorRequestReasonPdcSignalFingerprint,
  MonitorRequestReasonDirectedDrips,
  MonitorRequestReasonMax
} POWER_MONITOR_REQUEST_REASON;
```

Constants

MonitorRequestReasonUnknown	
MonitorRequestReasonPowerButton	
MonitorRequestReasonRemoteConnection	
MonitorRequestReasonScMonitorpower	
MonitorRequestReasonUserInput	
MonitorRequestReasonAcDcDisplayBurst	
MonitorRequestReasonUserDisplayBurst	
MonitorRequestReasonPoSetSystemState	
MonitorRequestReasonSetThreadExecutionState	
MonitorRequestReasonFullWake	
MonitorRequestReasonSessionUnlock	
MonitorRequestReasonScreenOffRequest	
MonitorRequestReasonIdleTimeout	
MonitorRequestReasonPolicyChange	
MonitorRequestReasonSleepButton	
MonitorRequestReasonLid	
MonitorRequestReasonBatteryCountChange	
MonitorRequestReasonGracePeriod	
MonitorRequestReasonPnP	
MonitorRequestReasonDP	
MonitorRequestReasonSxTransition	
MonitorRequestReasonSystemIdle	
MonitorRequestReasonNearProximity	
MonitorRequestReasonThermalStandby	
MonitorRequestReasonResumePdc	

MonitorRequestReasonResumeS4	
MonitorRequestReasonTerminal	
MonitorRequestReasonPdcSignal	
MonitorRequestReasonAcDcDisplayBurstSuppressed	
MonitorRequestReasonSystemStateEntered	
MonitorRequestReasonWinrt	
MonitorRequestReasonUserInputKeyboard	
MonitorRequestReasonUserInputMouse	
MonitorRequestReasonUserInputTouch	
MonitorRequestReasonUserInputPen	
MonitorRequestReasonUserInputAccelerometer	
MonitorRequestReasonUserInputHid	
MonitorRequestReasonUserInputPoUserPresent	
MonitorRequestReasonUserInputSessionSwitch	
MonitorRequestReasonUserInputInitialization	
MonitorRequestReasonPdcSignalWindowsMobilePwrNotif	
MonitorRequestReasonPdcSignalWindowsMobileShell	
MonitorRequestReasonPdcSignalHeyCortana	
MonitorRequestReasonPdcSignalHolographicShell	
MonitorRequestReasonPdcSignalFingerprint	
MonitorRequestReasonDirectedDrips	
MonitorRequestReasonMax	

Requirements

Header	ntpoapi.h

COUNTED_REASON_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **COUNTED_REASON_CONTEXT** structure contains one or more strings that give reasons for a power request.

Syntax

```
typedef struct _COUNTED_REASON_CONTEXT {
  ULONG Version;
  ULONG Flags;
  union {
    struct {
      UNICODE_STRING  ResourceFileName;
      USHORT          ResourceReasonId;
      ULONG           StringCount;
      PUNICODE_STRING ReasonStrings;
    } DUMMYSTRUCTNAME;
    UNICODE_STRING SimpleString;
  } DUMMYUNIONNAME;
} COUNTED_REASON_CONTEXT, *PCOUNTED_REASON_CONTEXT;
```

Members

`Version`

The version number of the structure. Set this member to DIAGNOSTIC_REASON_VERSION.

`Flags`

Indicates whether the structure contains a simple reason string or a detailed set of reason strings. Set this member to one of the following constants:

- DIAGNOSTIC_REASON_SIMPLE_STRING
- DIAGNOSTIC_REASON_DETAILED_STRING

If **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING, the **SimpleString** member of the union is valid. If **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING, the **ResourceFileName**, **ResourceReasonId**, **StringCount**, and **ReasonStrings** members are valid (and the **SimpleString** member is not valid).

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ResourceFileName`

A pointer to a wide-character, null-terminated string that contains the pathname of a resource file. This resource file contains one or more localized strings that give reasons for a power request. This member is optional and can be specified as **NULL** or as an empty string if no resource file is required. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ResourceReasonId`

The resource ID assigned to the first reason string in the resource file that is specified by **ResourceFileName**. This

member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.StringCount`

The number of reason strings in the **ReasonStrings** array or in the resource file that is specified by **ResourceFileName**. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReasonStrings`

A pointer to an array of string pointers. Each array element is a pointer to a wide-character, null-terminated string. The number of array elements is specified by **StringCount**. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.SimpleString`

A pointer to a wide-character, null-terminated string that explains the reason for a power request. This member is valid only if **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING.

Remarks

This structure is used by the PoCreatePowerRequest routine.

The power manager uses the reason string or strings contained in this structure as a diagnostic aid during functional and performance testing.

The **COUNTED_REASON_CONTEXT** structure can contain either a simple reason string or a set of detailed reason strings. If **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING, the **SimpleString** member points to a string that explains the reason for the power request. If **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING, the **ResourceFileName**, **ResourceReasonId**, **StringCount**, and **ReasonStrings** members can give a detailed set of reasons for the power request.

The DIAGNOSTIC_REASON_DETAILED_STRING flag supports localization. If the localized resource file specified by **ResourceFileName** exists, the power manager retrieves a set of resource strings from the file and uses these strings in place of the strings contained in the **ReasonStrings** array. **ResourceId** specifies the resource index of the first string in the file to use as a reason string, and **StringCount** specifies the number of resource strings to use as reason strings. The resource strings must have consecutive resource indexes.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

PoCreatePowerRequest

DEVICE_POWER_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_POWER_STATE** enumeration type indicates a device power state.

Syntax

```
typedef enum _DEVICE_POWER_STATE {
  PowerDeviceUnspecified,
  PowerDeviceD0,
  PowerDeviceD1,
  PowerDeviceD2,
  PowerDeviceD3,
  PowerDeviceMaximum
} DEVICE_POWER_STATE, *PDEVICE_POWER_STATE;
```

Constants

PowerDeviceUnspecified	Indicates an unspecified device power state.
PowerDeviceD0	Indicates a maximum device power state, which corresponds to device working state D0.
PowerDeviceD1	Indicates a device sleeping state less than **PowerDeviceD0** and greater than **PowerDeviceD2**, which corresponds to device power state D1.
PowerDeviceD2	Indicates a device sleeping state less than **PowerDeviceD1** and greater than **PowerDeviceD3**, which corresponds to device power state D2.
PowerDeviceD3	Indicates the lowest-powered device sleeping state, which corresponds to device power state D3.
PowerDeviceMaximum	The number of device power state values for this enumeration type that represent actual power states. The value of the other device power states is less than this value.

Remarks

A power state indicates the level of power consumption—and thus the extent of computing activity—by the system or by a single device. The power manager sets the power state of the system as a whole, where the system power state is indicated by one of the values of the SYSTEM_POWER_STATE enumeration type. Device drivers set the power state of their individual devices, where the device power state is indicated by one of the values of the **DEVICE_POWER_STATE** enumeration type.

For more information about system power states, see Handling System Power State Requests. For more information about device power states, see Managing Power for Individual Devices. For more information about power management in general, see Introduction to Power Management.

Requirements

Header	ntpoapi.h (include Wdm.h, Ntpoapi.h)

See Also

SYSTEM_POWER_STATE

MONITOR_DISPLAY_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Indicates the power state of the monitor being displayed on.

Syntax

```
typedef enum _MONITOR_DISPLAY_STATE {
  PowerMonitorOff,
  PowerMonitorOn,
  PowerMonitorDim
} MONITOR_DISPLAY_STATE, *PMONITOR_DISPLAY_STATE;
```

Constants

PowerMonitorOff	This indicates that the monitor is off.
PowerMonitorOn	This indicates that the monitor is on.
PowerMonitorDim	This indicates that the monitor is dim.

Requirements

Header	ntpoapi.h (include Ntpoapi.h)

PO_SPR_ACTIVE_SESSION_DATA structure

1/25/2019 • 2 minutes to read • Edit Online

Stores information that indicates whether a System Power Report's Active Session is starting or ending.

Syntax

```
typedef struct _PO_SPR_ACTIVE_SESSION_DATA {
  BOOLEAN Start;
  GUID    ActiveSessionGuid;
} PO_SPR_ACTIVE_SESSION_DATA, *PPO_SPR_ACTIVE_SESSION_DATA;
```

Members

`Start`

A boolean value that indicates whether the session is starting or ending.

`ActiveSessionGuid`

The Active session GUID for the session.

Remarks

To register for System Power Report's Active Session notifications, you must call **PowerSettingRegisterNotification** function with these these values:

- *SettingGuid*: Pass GUID_SPR_ACTIVE_SESSION_CHANGE.
- *Flags*: Set to DEVICE_NOTIFY_CALLBACK.
- *Recipient*: A **DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS** structure, in which the **Callback** member is set to your implementation of the DEVICE_NOTIFY_CALLBACK_ROUTINE callback routine that gets called when the session starts or ends.

If the registration is successful, the DEVICE_NOTIFY_CALLBACK_ROUTINE callback is invoked with these value:

- *Type*: PBT_POWERSETTINGCHANGE to indicate a power setting change.
- *UpdateSetting*: Is a **POWERBROADCAST_SETTING** structure; the **PowerSetting** member is set to GUID_SPR_ACTIVE_SESSION_CHANGE, the **DataLength** member is a **_PO_SPR_ACTIVE_SESSION_DATA** structure.

Requirements

Header	ntpoapi.h

See Also

PowerSettingRegisterNotification

POWERBROADCAST_SETTING

POWER_PLATFORM_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_PLATFORM_INFORMATION** structure contains information about the power capabilities of the system.

Syntax

```
typedef struct _POWER_PLATFORM_INFORMATION {
  BOOLEAN AoAc;
} POWER_PLATFORM_INFORMATION, *PPOWER_PLATFORM_INFORMATION;
```

Members

`AoAc`

Indicates whether the system supports the connected standby power model.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Header	ntpoapi.h (include Ntpoapi.h)

POWER_REQUEST_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_REQUEST_TYPE** enumeration indicates the power request type.

Syntax

```
typedef enum _POWER_REQUEST_TYPE {
  PowerRequestDisplayRequired,
  PowerRequestSystemRequired,
  PowerRequestAwayModeRequired,
  PowerRequestExecutionRequired
} POWER_REQUEST_TYPE, *PPOWER_REQUEST_TYPE;
```

Constants

PowerRequestDisplayRequired	Not used by drivers. For more information, see Remarks.
PowerRequestSystemRequired	Prevents the computer from automatically entering sleep mode after a period of user inactivity.
PowerRequestAwayModeRequired	Not used by drivers. For more information, see Remarks.
PowerRequestExecutionRequired	Not used by drivers. For more information, see Remarks.

Remarks

This enumeration is used by the kernel-mode PoClearPowerRequest and PoSetPowerRequest routines. Drivers that call these routines must specify the **PowerRequestSystemRequired** enumeration value.

The other three enumeration values—**PowerRequestDisplayRequired**, **PowerRequestAwayModeRequired**, and **PowerRequestExecutionRequired**—are not used by drivers. Applications specify these power request types in calls to the PowerSetRequest and PowerClearRequest functions.

A **PowerRequestDisplayRequired** power request has the following effects:

- After a period of user inactivity, the session display stays on and will not automatically turn off. If the display is already turned off, the power request turns the display on.
- A screensaver will not automatically start after a period of user inactivity. If a screensaver is already running, the power request stops the screensaver.
- The session will not be automatically locked after a period of user inactivity. If the session is already locked when the driver sends the power request, the session remains locked.

While a **PowerRequestAwayModeRequired** power request is in effect, if the user tries to put the computer into sleep mode (for example, by clicking **Start** and then clicking **Sleep**), the power manager turns off audio and video so that the computer appears to be in sleep mode, but the computer continues to run.

While a **PowerRequestExecutionRequired** power request is in effect, the calling process continues to run instead of being suspended or terminated by process lifetime management (PLM) mechanisms. When and how long the

process is allowed to run depends on the operating system and power policy settings. This type of power request is supported starting with Windows 8.

Requirements

Minimum supported client	Supported starting with Windows 7.
Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

PoClearPowerRequest

PoSetPowerRequest

PowerClearRequest

PowerSetRequest

POWER_STATE union

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_STATE** union specifies a system power state value or a device power state value.

Syntax

```
typedef union _POWER_STATE {
  SYSTEM_POWER_STATE SystemState;
  DEVICE_POWER_STATE DeviceState;
} POWER_STATE, *PPOWER_STATE;
```

Members

`SystemState`

A system power state value of type SYSTEM_POWER_STATE.

`DeviceState`

A device power state value of type DEVICE_POWER_STATE.

Remarks

The **POWER_STATE** union is used in conjunction with the POWER_STATE_TYPE enumeration type to specify a system power state value or a device power state value. This union is also used in cases where the power state type is implicitly determined by the context in which it is used. For example, see PoRequestPowerIrp and PoSetPowerState.

For more information about power management, see Introduction to Power Management.

Requirements

Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

POWER_STATE_TYPE

PoRequestPowerIrp

PoSetPowerState

SYSTEM_POWER_STATE

POWER_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_STATE_TYPE** enumeration type indicates that a power state value is a system power state or a device power state.

Syntax

```
typedef enum _POWER_STATE_TYPE {
  SystemPowerState,
  DevicePowerState
} POWER_STATE_TYPE, *PPOWER_STATE_TYPE;
```

Constants

SystemPowerState	Indicates a SYSTEM_POWER_STATE value.
DevicePowerState	Indicates a DEVICE_POWER_STATE value.

Remarks

The **POWER_STATE_TYPE** enumeration type is used in conjunction with a value of type POWER_STATE to indicate that the power state value is a system power state value or a device power state value. For an example, see PoSetPowerState.

Requirements

Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

POWER_STATE

PoSetPowerState

SYSTEM_POWER_STATE

SYSTEM_POWER_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **SYSTEM_POWER_STATE** enumeration type is used to indicate a system power state.

Syntax

```
typedef enum _SYSTEM_POWER_STATE {
  PowerSystemUnspecified,
  PowerSystemWorking,
  PowerSystemSleeping1,
  PowerSystemSleeping2,
  PowerSystemSleeping3,
  PowerSystemHibernate,
  PowerSystemShutdown,
  PowerSystemMaximum
} SYSTEM_POWER_STATE, *PSYSTEM_POWER_STATE;
```

Constants

PowerSystemUnspecified	Indicates an unspecified system power state.
PowerSystemWorking	Indicates maximum system power, which corresponds to system working state S0.
PowerSystemSleeping1	Indicates a system sleeping state less than **PowerSystemWorking** and greater than **PowerSystemSleeping2**, which corresponds to system power state S1.
PowerSystemSleeping2	Indicates a system sleeping state less than **PowerSystemSleeping1** and greater than **PowerSystemSleeping3**, which corresponds to system power state S2.
PowerSystemSleeping3	Indicates a system sleeping state less than **PowerSystemSleeping2** and greater than **PowerSystemHibernate**, which corresponds to system power state S3.
PowerSystemHibernate	Indicates the lowest-powered sleeping state, which corresponds to system power state S4.
PowerSystemShutdown	Indicates the system is turned off, which corresponds to system shutdown state S5.
PowerSystemMaximum	The number of system power state values for this enumeration type that represents actual power states. This value is the number of elements in the **DeviceState** member of the DEVICE_CAPABILITIES structure for a device. The other system power state values are less than this value.

Remarks

A power state indicates the level of power consumption—and thus the extent of computing activity—by the system or by a single device. The power manager sets the power state of the system as a whole, where the system power state is indicated by one of the values of the **SYSTEM_POWER_STATE** enumeration type. Device drivers set the power state of their individual devices, where the device power state is indicated by one of the values of the DEVICE_POWER_STATE enumeration type.

For more information about system power states, see Handling System Power State Requests and for more information about device power states, see Managing Power for Individual Devices.

For more information about power management in general, see Introduction to Power Management.

Requirements

Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

SYSTEM_POWER_STATE_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **SYSTEM_POWER_STATE_CONTEXT** structure is a partially opaque system structure that contains information about the previous system power states of a computer.

Syntax

```
typedef struct _SYSTEM_POWER_STATE_CONTEXT {
  union {
    struct {
      ULONG Reserved1 : 8;
      ULONG TargetSystemState : 4;
      ULONG EffectiveSystemState : 4;
      ULONG CurrentSystemState : 4;
      ULONG IgnoreHibernationPath : 1;
      ULONG PseudoTransition : 1;
      ULONG KernelSoftReboot : 1;
      ULONG DirectedDripsTransition : 1;
      ULONG Reserved2 : 8;
    } DUMMYSTRUCTNAME;
    ULONG ContextAsUlong;
  } DUMMYUNIONNAME;
} SYSTEM_POWER_STATE_CONTEXT, *PSYSTEM_POWER_STATE_CONTEXT;
```

Members

`DUMMYUNIONNAME`

Unnamed union.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Reserved1`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.TargetSystemState`

The target system power state of the previous system power IRP that the driver received. This member is set to a SYSTEM_POWER_STATE enumeration value. Drivers should treat this member as read-only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EffectiveSystemState`

The effective previous system power state, as perceived by the user. This member is set to a **SYSTEM_POWER_STATE** enumeration value. Drivers should treat this member as read-only. This member value might not match the **TargetSystemState** member if, for example, the previous system power IRP indicated that the computer was about to enter hibernation, but a hybrid shutdown instead occurred to prepare the computer for a fast startup. For more information, see Remarks.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.CurrentSystemState`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IgnoreHibernationPath`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.PseudoTransition`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.KernelSoftReboot`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DirectedDripsTransition`

a Directed DRIPS transition.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Reserved2`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.ContextAsUlong`

Opaque member. Reserved for system use.

Remarks

Starting with Windows Vista, the I/O stack location in a system power IRP contains a **SYSTEM_POWER_STATE_CONTEXT** structure. The **Power** member of the **IO_STACK_LOCATION** structure contains a **SystemPowerStateContext** member, which is a **SYSTEM_POWER_STATE_CONTEXT** structure. For more information, see IO_STACK_LOCATION.

The size of the **SYSTEM_POWER_STATE_CONTEXT** structure is four bytes. This structure is divided into bit fields, most of which are opaque to drivers and reserved exclusively for use by the operating system. However, two of these bit fields, **TargetSystemState** and **EffectiveSystemState**, can be read by kernel-mode drivers to distinguish a fast startup from a wake-from-hibernation startup. For more information, see Distinguishing Fast Startup from Wake-from-Hibernation.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	ntpoapi.h (include Ntpoapi.h)

See Also

IO_STACK_LOCATION

SYSTEM_POWER_STATE

CM_POWER_DATA structure

1/25/2019 • 2 minutes to read • Edit Online

The **CM_POWER_DATA** structure contains information about a device's power management state and capabilities.

Syntax

```
typedef struct CM_Power_Data_s {
  ULONG              PD_Size;
  DEVICE_POWER_STATE PD_MostRecentPowerState;
  ULONG              PD_Capabilities;
  ULONG              PD_D1Latency;
  ULONG              PD_D2Latency;
  ULONG              PD_D3Latency;
  DEVICE_POWER_STATE PD_PowerStateMapping[POWER_SYSTEM_MAXIMUM];
  SYSTEM_POWER_STATE PD_DeepestSystemWake;
} CM_POWER_DATA, *PCM_POWER_DATA;
```

Members

`PD_Size`

The size, in bytes, of this structure.

`PD_MostRecentPowerState`

A DEVICE_POWER_STATE value representing the device's most recent power state.

`PD_Capabilities`

The device's power capabilities, represented by a combination of the following bit flags:

PDCAP_D0_SUPPORTED

PDCAP_D1_SUPPORTED

PDCAP_D2_SUPPORTED

PDCAP_D3_SUPPORTED

PDCAP_WAKE_FROM_D0_SUPPORTED

PDCAP_WAKE_FROM_D1_SUPPORTED

PDCAP_WAKE_FROM_D2_SUPPORTED

PDCAP_WAKE_FROM_D3_SUPPORTED

PDCAP_WARM_EJECT_SUPPORTED

`PD_D1Latency`

The device's latency when returning the D0 state from the D1 state. For more information, see the **D1Latency** member of DEVICE_CAPABILITIES.

`PD_D2Latency`

The device's latency when returning the D0 state from the D2 state. For more information, see the **D2Latency** member of **DEVICE_CAPABILITIES**.

`PD_D3Latency`

The device's latency when returning the D0 state from the D3 state. For more information, see the **D3Latency** member of **DEVICE_CAPABILITIES**.

`PD_PowerStateMapping`

An array of **DEVICE_POWER_STATE** values representing the maximum device power state achievable for each system power state. For more information, see the **DeviceState** member of DEVICE_CAPABILITIES.

`PD_DeepestSystemWake`

Specifies the least-powered system state from which the device can wake the system. **PD_DeepestSystemWake** typically indicates one of the system sleeping states, S1, S2, or S3 (as specified by **PowerSystemSleeping1**, **PowerSystemSleeping2**, and **PowerSystemSleeping3**, respectively). Note, however, that some devices might be able to wake the system from the system hibernate state, S4 (as specified by **PowerSystemHibernate**), or even from the system shutdown state, S5 (as specified by **PowerSystemShutdown**).

Requirements

Minimum supported client	Available in Microsoft Windows XP and later versions of the Windows operating system.
Header	ntpoapi.h (include Ntddk.h, Ntpoapi.h)

See Also

DEVICE_CAPABILITIES

DEVICE_POWER_STATE

SetupDiGetDeviceRegistryProperty

NtPowerInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPowerInformation** routine sets or retrieves system power information.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPowerInformation(
  POWER_INFORMATION_LEVEL InformationLevel,
  PVOID                   InputBuffer,
  ULONG                   InputBufferLength,
  PVOID                   OutputBuffer,
  ULONG                   OutputBufferLength
);
```

Parameters

`InformationLevel`

Specifies the requested information level, which indicates the specific power information to be set or retrieved. Currently, the only supported *POWER_INFORMATION_LEVEL* value is **PlatformInformation**.

VALUE	MEANING
PlatformInformation	Information represents the currently supported power capabilities of the system. Information may change as drivers are installed. For example, the installation of legacy device drivers that do not support power management might modify the capabilities of the system.

`InputBuffer`

Pointer to a caller-allocated input buffer. This parameter must be **NULL**, otherwise **ERROR_INVALID_PARAMETER** is returned.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. The parameter must be set to zero.

`OutputBuffer`

A pointer to an output buffer. The data type of this buffer depends on the information level requested in the *InformationLevel* parameter. For the **PlatformInformation** level, the only currently supported value, the *OutputBuffer* parameter is required and should be of the POWER_PLATFORM_INFORMATION type.

`OutputBufferLength`

Size, in bytes, of the output buffer. Depending on the information level requested, the buffer may be variably sized. *PlatformInformation*, the only currently supported value, requires a buffer that is the size of a POWER_PLATFORM_INFORMATION structure.

Return Value

Returns STATUS_SUCCESS if the call is successful. If the call fails, possible error codes include the following:

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The output buffer is of insufficient size to contain the data being returned.
STATUS_INVALID_PARAMETER	The **PlatformInformation** information level, which is the only currently supported value, requires no input buffer and must contain an output buffer. The caller either supplied an input buffer or no output buffer.
STATUS_ACCESS_DENIED	The caller had insufficient access rights to perform the requested action.

Remarks

NtPowerInformation and **ZwPowerInformation** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Examples

This example illustrates a valid function call.

C++

```
POWER_PLATFORM_INFORMATION PlatformInfo = {0};
NTSTATUS Result = NtPowerInformation(PlatformInformation, NULL, 0, &PlatformInfo, sizeof(PlatformInfo));
```

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Target Platform	Universal
Header	ntpoapi.h (include Wdm.h, Ntddk.h, Ntpoapi.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

POWER_PLATFORM_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ntstrsafe.h header

1/8/2019 • 8 minutes to read • Edit Online

If you use the safe string functions instead of the string manipulation functions that are provided by C-language run-time libraries, you protect your code from buffer overrun errors that can make code untrustworthy. This header declares functions that copy, concatenate, and format strings in a manner that prevents buffer overrun errors.

To manipulate Unicode and ANSI strings, each function is available in two versions:

- A W-suffixed version that supports two-byte Unicode characters.
- An A-suffixed version that supports one-byte ANSI characters.

Unicode strings are specified within **UNICODE_STRING** structures in kernel-mode drivers.

For more information, see:

- Using Safe String Functions ntstrsafe.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
RtlStringCbCatA	The RtlStringCbCatW and RtlStringCbCatA functions concatenate two byte-counted strings.
RtlStringCbCatExA	The RtlStringCbCatExW and RtlStringCbCatExA functions concatenate two byte-counted strings.
RtlStringCbCatExW	The RtlStringCbCatExW and RtlStringCbCatExA functions concatenate two byte-counted strings.
RtlStringCbCatNA	The RtlStringCbCatNW and RtlStringCbCatNA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNExA	The RtlStringCbCatNExW and RtlStringCbCatNExA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNExW	The RtlStringCbCatNExW and RtlStringCbCatNExA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatNW	The RtlStringCbCatNW and RtlStringCbCatNA functions concatenate two byte-counted strings while limiting the size of the appended string.
RtlStringCbCatW	The RtlStringCbCatW and RtlStringCbCatA functions concatenate two byte-counted strings.
RtlStringCbCopyA	The RtlStringCbCopyW and RtlStringCbCopyA functions copy a byte-counted string into a buffer.

TITLE	DESCRIPTION
RtlStringCbCopyExA	The RtlStringCbCopyExW and RtlStringCbCopyExA functions copy a byte-counted string into a buffer.
RtlStringCbCopyExW	The RtlStringCbCopyExW and RtlStringCbCopyExA functions copy a byte-counted string into a buffer.
RtlStringCbCopyNA	The RtlStringCbCopyNW and RtlStringCbCopyNA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNExA	The RtlStringCbCopyNExW and RtlStringCbCopyNExA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNExW	The RtlStringCbCopyNExW and RtlStringCbCopyNExA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyNW	The RtlStringCbCopyNW and RtlStringCbCopyNA functions copy a byte-counted string to a buffer while limiting the size of the copied string.
RtlStringCbCopyUnicodeString	The RtlStringCbCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCbCopyUnicodeStringEx	The RtlStringCbCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCbCopyW	The RtlStringCbCopyW and RtlStringCbCopyA functions copy a byte-counted string into a buffer.
RtlStringCbLengthA	The RtlStringCbLengthW and RtlStringCbLengthA functions determine the length, in bytes, of a supplied string.
RtlStringCbLengthW	The RtlStringCbLengthW and RtlStringCbLengthA functions determine the length, in bytes, of a supplied string.
RtlStringCbPrintfA	The RtlStringCbPrintfW and RtlStringCbPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfExA	The RtlStringCbPrintfExW and RtlStringCbPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfExW	The RtlStringCbPrintfExW and RtlStringCbPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbPrintfW	The RtlStringCbPrintfW and RtlStringCbPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.

TITLE	DESCRIPTION
RtlStringCbVPrintfA	The RtlStringCbVPrintfW and RtlStringCbVPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfExA	The RtlStringCbVPrintfExW and RtlStringCbVPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfExW	The RtlStringCbVPrintfExW and RtlStringCbVPrintfExA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCbVPrintfW	The RtlStringCbVPrintfW and RtlStringCbVPrintfA functions create a byte-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchCatA	The RtlStringCchCatW and RtlStringCchCatA functions concatenate two character-counted strings.
RtlStringCchCatExA	The RtlStringCchCatExW and RtlStringCchCatExA functions concatenate two character-counted strings.
RtlStringCchCatExW	The RtlStringCchCatExW and RtlStringCchCatExA functions concatenate two character-counted strings.
RtlStringCchCatNA	The RtlStringCchCatNW and RtlStringCchCatNA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNExA	The RtlStringCchCatNExW and RtlStringCchCatNExA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNExW	The RtlStringCchCatNExW and RtlStringCchCatNExA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatNW	The RtlStringCchCatNW and RtlStringCchCatNA functions concatenate two character-counted strings while limiting the size of the appended string.
RtlStringCchCatW	The RtlStringCchCatW and RtlStringCchCatA functions concatenate two character-counted strings.
RtlStringCchCopyA	The RtlStringCchCopyW and RtlStringCchCopyA functions copy a null-terminated source string into a destination buffer of specified length.
RtlStringCchCopyExA	The RtlStringCchCopyExW and RtlStringCchCopyExA functions copy a character-counted string into a buffer.
RtlStringCchCopyExW	The RtlStringCchCopyExW and RtlStringCchCopyExA functions copy a character-counted string into a buffer.

TITLE	DESCRIPTION
RtlStringCchCopyNA	The RtlStringCchCopyNW and RtlStringCchCopyNA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNExA	The RtlStringCchCopyNExW and RtlStringCchCopyNExA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNExW	The RtlStringCchCopyNExW and RtlStringCchCopyNExA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyNW	The RtlStringCchCopyNW and RtlStringCchCopyNA functions copy a character-counted string to a buffer while limiting the size of the copied string.
RtlStringCchCopyUnicodeString	The RtlStringCchCopyUnicodeString function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyUnicodeStringEx	The RtlStringCchCopyUnicodeStringEx function copies the contents of a UNICODE_STRING structure to a specified destination.
RtlStringCchCopyW	The RtlStringCchCopyW and RtlStringCchCopyA functions copy a null-terminated source string into a destination buffer of specified length.
RtlStringCchLengthA	The RtlStringCchLengthW and RtlStringCchLengthA functions determine the length, in characters, of a supplied string.
RtlStringCchLengthW	The RtlStringCchLengthW and RtlStringCchLengthA functions determine the length, in characters, of a supplied string.
RtlStringCchPrintfA	The RtlStringCchPrintfW and RtlStringCchPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfExA	The RtlStringCchPrintfExW and RtlStringCchPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfExW	The RtlStringCchPrintfExW and RtlStringCchPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchPrintfW	The RtlStringCchPrintfW and RtlStringCchPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfA	The RtlStringCchVPrintfW and RtlStringCchVPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.

TITLE	DESCRIPTION
RtlStringCchVPrintfExA	The RtlStringCchVPrintfExW and RtlStringCchVPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfExW	The RtlStringCchVPrintfExW and RtlStringCchVPrintfExA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlStringCchVPrintfW	The RtlStringCchVPrintfW and RtlStringCchVPrintfA functions create a character-counted text string, with formatting that is based on supplied formatting information.
RtlUnalignedStringCbLengthW	The RtlUnalignedStringCbLengthW function is a version of the RtlStringCbLength function that accepts an unaligned pointer to a string of Unicode characters.
RtlUnalignedStringCchLengthW	The RtlUnalignedStringCchLengthW function is a version of the RtlStringCchLength function that accepts an unaligned pointer to a string of Unicode characters.
RtlUnicodeStringCat	The RtlUnicodeStringCat function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatEx	The RtlUnicodeStringCatEx function concatenates two strings that are contained in UNICODE_STRING structures.
RtlUnicodeStringCatString	The RtlUnicodeStringCatString function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCatStringEx	The RtlUnicodeStringCatStringEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.
RtlUnicodeStringCbCatN	The RtlUnicodeStringCbCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCbCatNEx	The RtlUnicodeStringCbCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCbCatStringN	The RtlUnicodeStringCbCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCbCatStringNEx	The RtlUnicodeStringCbCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCbCopyN	The RtlUnicodeStringCbCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.

TITLE	DESCRIPTION
RtlUnicodeStringCbCopyNEx	The RtlUnicodeStringCbCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringN	The RtlUnicodeStringCbCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCbCopyStringNEx	The RtlUnicodeStringCbCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCchCatN	The RtlUnicodeStringCchCatN function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatNEx	The RtlUnicodeStringCchCatNEx function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.
RtlUnicodeStringCchCatStringN	The RtlUnicodeStringCchCatStringN function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCatStringNEx	The RtlUnicodeStringCchCatStringNEx function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.
RtlUnicodeStringCchCopyN	The RtlUnicodeStringCchCopyN function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyNEx	The RtlUnicodeStringCchCopyNEx function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.
RtlUnicodeStringCchCopyStringN	The RtlUnicodeStringCchCopyStringN function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCchCopyStringNEx	The RtlUnicodeStringCchCopyStringNEx function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.
RtlUnicodeStringCopy	The RtlUnicodeStringCopy function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyEx	The RtlUnicodeStringCopyEx function copies a string from one UNICODE_STRING structure to another.
RtlUnicodeStringCopyString	The RtlUnicodeStringCopyString function copies a string into a UNICODE_STRING structure.

TITLE	DESCRIPTION
RtlUnicodeStringCopyStringEx	The RtlUnicodeStringCopyStringEx function copies a string into a UNICODE_STRING structure.
RtlUnicodeStringInit	The RtlUnicodeStringInit function initializes a UNICODE_STRING structure.
RtlUnicodeStringInitEx	The RtlUnicodeStringInitEx function initializes a UNICODE_STRING structure.
RtlUnicodeStringPrintf	The RtlUnicodeStringPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringPrintfEx	The RtlUnicodeStringPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringValidate	The RtlUnicodeStringValidate function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringValidateEx	The RtlUnicodeStringValidateEx function validates the contents of a UNICODE_STRING structure.
RtlUnicodeStringVPrintf	The RtlUnicodeStringVPrintf function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.
RtlUnicodeStringVPrintfEx	The RtlUnicodeStringVPrintfEx function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

RtlStringCbCatA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCatW** and **RtlStringCbCatA** functions concatenate two byte-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character.

`cbDest`

The size of the destination buffer, in bytes. The destination buffer must be large enough to include the concatenated strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The STATUS_INVALID_PARAMETER return value means that one of the following occurred: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatW and **RtlStringCbCatA** should be used instead of the following functions:

- **strcat**
- **wcscat**

Because **RtlStringCbCatW** and **RtlStringCbCatA** receive the size of the destination buffer as input, they will not write past the end of the buffer.

Use **RtlStringCbCatW** to handle Unicode strings and **RtlStringCbCatA** to handle ANSI strings. The form to use is determined by your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatW**
char	"string"	**RtlStringCbCatA**

If *pszSrc* and *pszDest* point to strings that overlap, the behavior of the functions is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCbCatExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCatExW** and **RtlStringCbCatExA** functions concatenate two byte-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszSrc,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the strings were fully concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER when: • An invalid flag was specified. • The value in cbDest is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatExW and **RtlStringCbCatExA** should be used instead of the following functions:

- **strcat**
- **wcscat**

Because **RtlStringCbCatExW** and **RtlStringCbCatExA** receive the size of the destination buffer as input, they will not write past the end of the buffer.

RtlStringCbCatEx adds to the functionality of RtlStringCbCat by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can also be passed to the function for additional control.

Use **RtlStringCbCatExW** to handle Unicode strings and **RtlStringCbCatExA** to handle ANSI strings. The form to use is determined by your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatExW**
char	"string"	**RtlStringCbCatExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCbCatExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCatExW** and **RtlStringCbCatExA** functions concatenate two byte-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszSrc,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcbRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the strings were fully concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER when: • An invalid flag was specified. • The value in cbDest is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatExW and **RtlStringCbCatExA** should be used instead of the following functions:

- **strcat**
- **wcscat**

Because **RtlStringCbCatExW** and **RtlStringCbCatExA** receive the size of the destination buffer as input, they will not write past the end of the buffer.

RtlStringCbCatEx adds to the functionality of RtlStringCbCat by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can also be passed to the function for additional control.

Use **RtlStringCbCatExW** to handle Unicode strings and **RtlStringCbCatExA** to handle ANSI strings. The form to use is determined by your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatExW**
char	"string"	**RtlStringCbCatExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCbCatNA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCatNW** and **RtlStringCbCatNA** functions concatenate two byte-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatNA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cbDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cbToAppend
);
```

Parameters

`pszDest`

Pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cbMaxAppend* bytes, is added to the end of the string at *pszDest* and terminated with a null character.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

Pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

`cbToAppend`

The maximum number of bytes to append to the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatNW and **RtlStringCbCatNA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCatNW** and **RtlStringCbCatNA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCbCatNW** to handle Unicode strings and **RtlStringCbCatNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatNW**
char	"string"	**RtlStringCbCatNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCbCatNEx

RtlStringCchCatN

RtlStringCbCatNExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCatNExW** and **RtlStringCbCatNExA** functions concatenate two byte-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatNExA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cbDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cbToAppend,
  NTSTRSAFE_PSTR *ppszDestEnd,
  size_t         *pcbRemaining,
  DWORD          dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cbMaxAppend* bytes, is added to the end of the string at *pszDest*, and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToAppend`

The maximum number of bytes to append to *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes

that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how

to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were fully concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatNExW and **RtlStringCbCatNExA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCatNExW** and **RtlStringCbCatNExA** to ensure that they do not write past the end of the buffer.

RtlStringCbCatNEx adds to the functionality of RtlStringCbCatN by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbCatNExW** to handle Unicode strings and **RtlStringCbCatNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatNExW**

char	"string"	**RtlStringCbCatNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCbCatN

RtlStringCchCatNEx

RtlStringCbCatNExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCatNExW** and **RtlStringCbCatNExA** functions concatenate two byte-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatNExW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cbDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cbToAppend,
  NTSTRSAFE_PWSTR *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cbMaxAppend* bytes, is added to the end of the string at *pszDest*, and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToAppend`

The maximum number of bytes to append to *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes

that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how

to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were fully concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatNExW and **RtlStringCbCatNExA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCatNExW** and **RtlStringCbCatNExA** to ensure that they do not write past the end of the buffer.

RtlStringCbCatNEx adds to the functionality of RtlStringCbCatN by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbCatNExW** to handle Unicode strings and **RtlStringCbCatNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatNExW**

char	"string"	**RtlStringCbCatNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCbCatN

RtlStringCchCatNEx

RtlStringCbCatNW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCatNW** and **RtlStringCbCatNA** functions concatenate two byte-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatNW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cbDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cbToAppend
);
```

Parameters

`pszDest`

Pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cbMaxAppend* bytes, is added to the end of the string at *pszDest* and terminated with a null character.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to include both strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

Pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

`cbToAppend`

The maximum number of bytes to append to the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatNW and **RtlStringCbCatNA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCatNW** and **RtlStringCbCatNA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCbCatNW** to handle Unicode strings and **RtlStringCbCatNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatNW**
char	"string"	**RtlStringCbCatNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCbCatNEx

RtlStringCchCatN

RtlStringCbCatW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCatW** and **RtlStringCbCatA** functions concatenate two byte-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCbCatW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character.

`cbDest`

The size of the destination buffer, in bytes. The destination buffer must be large enough to include the concatenated strings and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The STATUS_INVALID_PARAMETER return value means that one of the following occurred: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCatW and **RtlStringCbCatA** should be used instead of the following functions:

- **strcat**
- **wcscat**

Because **RtlStringCbCatW** and **RtlStringCbCatA** receive the size of the destination buffer as input, they will not write past the end of the buffer.

Use **RtlStringCbCatW** to handle Unicode strings and **RtlStringCbCatA** to handle ANSI strings. The form to use is determined by your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCatW**
char	"string"	**RtlStringCbCatA**

If *pszSrc* and *pszDest* point to strings that overlap, the behavior of the functions is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCbCopyA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCopyW** and **RtlStringCbCopyA** functions copy a byte-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyA and **RtlStringCbCopyW** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

RtlStringCbCopyA and **RtlStringCbCopyW** are not replacements for **strncpy**. To replace **strncpy**, use RtlStringCbCopyN or RtlStringCbCopyNEx.

The size of the destination buffer is provided to the function to ensure that **RtlStringCbCopy** does not write past the end of the buffer.

Use **RtlStringCbCopyW** to handle Unicode strings and **RtlStringCbCopyA** to handle ANSI strings. The form you use depends on your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyW**
char	"string"	**RtlStringCbCopyA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCopyEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyEx

RtlStringCchCopy

RtlStringCbCopyExA function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCbCopyExW** and **RtlStringCbCopyExA** functions copy a byte-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszSrc,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyExA and **RtlStringCbCopyExW** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCopyExA** and **RtlStringCbCopyExW** to ensure that they do not write past the end of the buffer.

RtlStringCbCopyEx adds to the functionality of RtlStringCbCopy by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbCopyExW** to handle Unicode strings and **RtlStringCbCopyExA** to handle ANSI strings. The form you use depends on your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyExW**
char	"string"	**RtlStringCbCopyExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCchCopyEx

RtlStringCbCopyExW function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCbCopyExW** and **RtlStringCbCopyExA** functions copy a byte-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszSrc,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcbRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyExA and **RtlStringCbCopyExW** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

The size, in bytes, of the destination buffer is provided to **RtlStringCbCopyExA** and **RtlStringCbCopyExW** to ensure that they do not write past the end of the buffer.

RtlStringCbCopyEx adds to the functionality of RtlStringCbCopy by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbCopyExW** to handle Unicode strings and **RtlStringCbCopyExA** to handle ANSI strings. The form you use depends on your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyExW**
char	"string"	**RtlStringCbCopyExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCchCopyEx

RtlStringCbCopyNA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCopyNW** and **RtlStringCbCopyNA** functions copy a byte-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyNA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cbDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cbToCopy
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc*, up to *cbSrc* bytes, is copied to the buffer at *pszDest* and terminated with a null character.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for both the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * **sizeof**(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * **sizeof**(**char**).

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cbToCopy`

The maximum number of bytes to copy from *pszSrc* to *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyNW and **RtlStringCbCopyNA** should be used instead of **strncpy**. However, these functions differ in behavior. If *cbSrc* is larger than the number of bytes in *pszSrc*, the **RtlStringCbCopyN** functions—unlike **strncpy**—do not fill *pszDest* with null characters until *cbSrc* bytes have been copied.

RtlStringCbCopyN copies a given number of bytes from a source string. The size, in bytes, of the destination buffer is provided to the function to ensure that **RtlStringCbCopyN** does not write past the end of this buffer.

Use **RtlStringCbCopyNW** to handle Unicode strings and **RtlStringCbCopyNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyNW**
char	"string"	**RtlStringCbCopyNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, see RtlStringCbCopyNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCbCopyNEx

RtlStringCchCopyN

RtlStringCbCopyNExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCopyNExW** and **RtlStringCbCopyNExA** functions copy a byte-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyNExA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cbDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cbToCopy,
  NTSTRSAFE_PSTR *ppszDestEnd,
  size_t         *pcbRemaining,
  DWORD          dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToCopy`

The maximum number of bytes to copy from *pszSrc* to *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including those bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyNExW and **RtlStringCbCopyNExA** should be used instead of **strncpy**. However, these functions differ in behavior. If *cbSrc* is larger than the number of bytes in *pszSrc*, the **RtlStringCbCopyNEx** functions, unlike **strncpy**, do not fill *pszDest* with null characters until *cbSrc* bytes have been copied.

The size, in bytes, of the destination buffer is provided to **RtlStringCbCopyNExW** and **RtlStringCbCopyNExA** to ensure that they do not write past the end of this buffer.

RtlStringCbCopyNEx adds to the functionality of RtlStringCbCopyN by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags may also be passed to the function for additional control.

Use **RtlStringCbCopyNExW** to handle Unicode strings and **RtlStringCbCopyNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyNExW**
char	"string"	**RtlStringCbCopyNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyN

RtlStringCchCopyNEx

RtlStringCbCopyNExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbCopyNExW** and **RtlStringCbCopyNExA** functions copy a byte-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyNExW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cbDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cbToCopy,
  NTSTRSAFE_PWSTR *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToCopy`

The maximum number of bytes to copy from *pszSrc* to *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including those bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyNExW and **RtlStringCbCopyNExA** should be used instead of **strncpy**. However, these functions differ in behavior. If *cbSrc* is larger than the number of bytes in *pszSrc*, the **RtlStringCbCopyNEx** functions, unlike **strncpy**, do not fill *pszDest* with null characters until *cbSrc* bytes have been copied.

The size, in bytes, of the destination buffer is provided to **RtlStringCbCopyNExW** and **RtlStringCbCopyNExA** to ensure that they do not write past the end of this buffer.

RtlStringCbCopyNEx adds to the functionality of RtlStringCbCopyN by returning a pointer to the end of the destination string as well as the number of bytes left unused in that string. Flags may also be passed to the function for additional control.

Use **RtlStringCbCopyNExW** to handle Unicode strings and **RtlStringCbCopyNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyNExW**
char	"string"	**RtlStringCbCopyNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyN

RtlStringCchCopyNEx

RtlStringCbCopyNW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCopyNW** and **RtlStringCbCopyNA** functions copy a byte-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyNW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cbDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cbToCopy
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc*, up to *cbSrc* bytes, is copied to the buffer at *pszDest* and terminated with a null character.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for both the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * **sizeof**(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * **sizeof**(**char**).

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cbToCopy`

The maximum number of bytes to copy from *pszSrc* to *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyNW and **RtlStringCbCopyNA** should be used instead of **strncpy**. However, these functions differ in behavior. If *cbSrc* is larger than the number of bytes in *pszSrc*, the **RtlStringCbCopyN** functions—unlike **strncpy**—do not fill *pszDest* with null characters until *cbSrc* bytes have been copied.

RtlStringCbCopyN copies a given number of bytes from a source string. The size, in bytes, of the destination buffer is provided to the function to ensure that **RtlStringCbCopyN** does not write past the end of this buffer.

Use **RtlStringCbCopyNW** to handle Unicode strings and **RtlStringCbCopyNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyNW**
char	"string"	**RtlStringCbCopyNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, see RtlStringCbCopyNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCbCopyNEx

RtlStringCchCopyN

RtlStringCbCopyUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCopyUnicodeString** function copies the contents of a UNICODE_STRING structure to a specified destination.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyUnicodeString(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  PCUNICODE_STRING SourceString
);
```

Parameters

`pszDest`

A pointer to a buffer that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer at *pszDest* and terminated with a null character.

`cbDest`

The size, in bytes, of the destination buffer that *pszDest* points to. The buffer must be large enough to contain the string and the terminating null character. The maximum number of bytes in the buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * **sizeof**(WCHAR).

`SourceString`

A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * **sizeof**(WCHAR).

Return Value

RtlStringCbCopyUnicodeString returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STA TUS _BU FFE R_O VER FLO W	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlStringCbCopyUnicodeString returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The value in *cbDest* is larger than the maximum buffer size.
- The destination buffer (which *pszDest* points to) is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlStringCbCopyUnicodeString** function uses the destination buffer's size (which the *cbDest* parameter specifies) to ensure that the copy operation does not write past the end of the buffer.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString*and *pszDest* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlStringCbCopyUnicodeStringEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyUnicodeStringEx

RtlStringCchCopyUnicodeString

UNICODE_STRING

RtlStringCbCopyUnicodeStringEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCbCopyUnicodeStringEx** function copies the contents of a UNICODE_STRING structure to a specified destination.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyUnicodeStringEx(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  PCUNICODE_STRING SourceString,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcbRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

Optional. A pointer to a buffer that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer at *pszDest* and terminated with a null character. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character. The maximum number of bytes in the buffer is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

Optional. If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcbRemaining`

Optional. If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer that *pszDest* points to, including bytes that are used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND_NULL

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.

STRSAFE_IGNORE_NULLS

If this flag is set, either the source or destination pointer, or both, can be **NULL**.
RtlStringCbCopyUnicodeStringEx treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

RtlStringCbCopyUnicodeStringEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlStringCbCopyUnicodeStringEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in dwFlags.
- The value in *cbDest* is larger than the maximum buffer size.
- The destination buffer (which *pszDest* points to) is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL** or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlStringCbCopyUnicodeStringEx** function uses the destination buffer's size (which the *cbDest* parameter specifies) to ensure that the copy operation does not write past the end of the buffer.

RtlStringCbCopyUnicodeStringEx adds to the functionality of the RtlStringCbCopyUnicodeString function by returning a pointer to the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to the function for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString*and *pszDest* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *pszDest* pointer is **NULL**, the *SourceString*pointer must be **NULL** or the UNICODE_STRING structure must describe an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyUnicodeString

RtlStringCchCopyUnicodeStringEx

UNICODE_STRING

RtlStringCbCopyW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbCopyW** and **RtlStringCbCopyA** functions copy a byte-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCbCopyW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character.

`cbDest`

The size, in bytes, of the destination buffer. The buffer must be large enough for the string and the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbCopyA and **RtlStringCbCopyW** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

RtlStringCbCopyA and **RtlStringCbCopyW** are not replacements for **strncpy**. To replace **strncpy**, use RtlStringCbCopyN or RtlStringCbCopyNEx.

The size of the destination buffer is provided to the function to ensure that **RtlStringCbCopy** does not write past the end of the buffer.

Use **RtlStringCbCopyW** to handle Unicode strings and **RtlStringCbCopyA** to handle ANSI strings. The form you use depends on your data as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbCopyW**
char	"string"	**RtlStringCbCopyA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbCopyEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyEx

RtlStringCchCopy

RtlStringCbLengthA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbLengthW** and **RtlStringCbLengthA** functions determine the length, in bytes, of a supplied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbLengthA(
  STRSAFE_PCNZCH psz,
  size_t         cbMax,
  size_t         *pcbLength
);
```

Parameters

`psz`

A pointer to a buffer that contains a null-terminated string, the length of which will be checked.

`cbMax`

The maximum number of bytes allowed in the buffer that is pointed to by *psz*, including the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pcbLength`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in bytes, of the string that is contained in the buffer. The length does not include the string's terminating null character.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means the string at *psz* was not **NULL**, and the length of the string (including the terminating null character) is less than or equal to *cbMax* characters.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means the value in *psz* is **NULL**, *cbMax* is larger than NTSTRSAFE_MAX_CCH * sizeof(TCHAR), or *psz* is longer than *cbMax*.

Remarks

RtlStringCbLengthW and **RtlStringCbLengthA** should be used instead of **strlen**. Use these functions to ensure that a string is not larger than a given length, in bytes. If that condition is met, **RtlStringCbLengthW** and **RtlStringCbLengthA** return the current length of the string in bytes, not including those bytes used for the terminating null character.

Use **RtlStringCbLengthW** to handle Unicode strings and **RtlStringCbLengthA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbLengthW**
char	"string"	**RtlStringCbLengthA**

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCchLength

RtlStringCbLengthW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbLengthW** and **RtlStringCbLengthA** functions determine the length, in bytes, of a supplied string.

Syntax

```
NTSTRSAFEDDI RtlStringCbLengthW(
  STRSAFE_PCNZWCH psz,
  size_t          cbMax,
  size_t          *pcbLength
);
```

Parameters

`psz`

A pointer to a buffer that contains a null-terminated string, the length of which will be checked.

`cbMax`

The maximum number of bytes allowed in the buffer that is pointed to by *psz*, including the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pcbLength`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in bytes, of the string that is contained in the buffer. The length does not include the string's terminating null character.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means the string at *psz* was not **NULL**, and the length of the string (including the terminating null character) is less than or equal to *cbMax* characters.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means the value in *psz* is **NULL**, *cbMax* is larger than NTSTRSAFE_MAX_CCH * sizeof(TCHAR), or *psz* is longer than *cbMax*.

Remarks

RtlStringCbLengthW and **RtlStringCbLengthA** should be used instead of **strlen**. Use these functions to ensure that a string is not larger than a given length, in bytes. If that condition is met, **RtlStringCbLengthW** and **RtlStringCbLengthA** return the current length of the string in bytes, not including those bytes used for the terminating null character.

Use **RtlStringCbLengthW** to handle Unicode strings and **RtlStringCbLengthA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbLengthW**
char	"string"	**RtlStringCbLengthA**

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCchLength

RtlStringCbPrintfA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbPrintfW** and **RtlStringCbPrintfA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbPrintfA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`arg4`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the output string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbPrintfW and **RtlStringCbPrintfA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments, interpret them, and create a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbPrintfW** and **RtlStringCbPrintfA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCbPrintfW** to handle Unicode strings and **RtlStringCbPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbPrintfW**
char	"string"	**RtlStringCbPrintfA**

If *pszDest* and *pszFormat* point to overlapping string or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Examples

The following example shows a basic use of **RtlStringCbPrintfW** using four arguments.

```
int const arraysize = 30;
WCHAR pszDest[arraysize];
size_t cbDest = arraysize * sizeof(WCHAR);

LPCWSTR pszFormat = L"%s %d + %d = %d.";
WCHAR* pszTxt = L"The answer is";

NTSTATUS status = RtlStringCbPrintfW(pszDest, cbDest, pszFormat, pszTxt, 1, 2, 3);
```

The resultant string is "The answer is 1 + 2 = 3." It is contained in the buffer at *pszDest*.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCbVPrintf

RtlStringCchPrintf

RtlStringCbPrintfExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbPrintfExW** and **RtlStringCbPrintfExA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbPrintfExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags,
  NTSTRSAFE_PCSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL***pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`arg7`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbPrintfExW and **RtlStringCbPrintfExA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments, interpret them, and create a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbPrintfExW** and **RtlStringCbPrintfExA** to ensure that they do not write past the end of the buffer.

RtlStringCbPrintfExW and **RtlStringCbPrintfExA** add to the functionality of RtlStringCbPrintf by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbPrintfExW** to handle Unicode strings and **RtlStringCbPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbPrintfExW**
char	"string"	**RtlStringCbPrintfExA**

If *pszDest* and *pszFormat*point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszFormat* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintfEx

RtlStringCchPrintf

RtlStringCchPrintfEx

RtlStringCbPrintfExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbPrintfExW** and **RtlStringCbPrintfExA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbPrintfExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcbRemaining,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL***pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`arg7`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbPrintfExW and **RtlStringCbPrintfExA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments, interpret them, and create a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbPrintfExW** and **RtlStringCbPrintfExA** to ensure that they do not write past the end of the buffer.

RtlStringCbPrintfExW and **RtlStringCbPrintfExA** add to the functionality of RtlStringCbPrintf by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCbPrintfExW** to handle Unicode strings and **RtlStringCbPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbPrintfExW**
char	"string"	**RtlStringCbPrintfExA**

If *pszDest* and *pszFormat*point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszFormat* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintfEx

RtlStringCchPrintf

RtlStringCchPrintfEx

RtlStringCbPrintfW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbPrintfW** and **RtlStringCbPrintfA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbPrintfW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`arg4`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the output string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbPrintfW and **RtlStringCbPrintfA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments, interpret them, and create a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbPrintfW** and **RtlStringCbPrintfA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCbPrintfW** to handle Unicode strings and **RtlStringCbPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbPrintfW**
char	"string"	**RtlStringCbPrintfA**

If *pszDest* and *pszFormat* point to overlapping string or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Examples

The following example shows a basic use of **RtlStringCbPrintfW** using four arguments.

```
int const arraysize = 30;
WCHAR pszDest[arraysize];
size_t cbDest = arraysize * sizeof(WCHAR);

LPCWSTR pszFormat = L"%s %d + %d = %d.";
WCHAR* pszTxt = L"The answer is";

NTSTATUS status = RtlStringCbPrintfW(pszDest, cbDest, pszFormat, pszTxt, 1, 2, 3);
```

The resultant string is "The answer is 1 + 2 = 3." It is contained in the buffer at *pszDest*.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCbVPrintf

RtlStringCchPrintf

RtlStringCbVPrintfA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbVPrintfW** and **RtlStringCbVPrintfA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbVPrintfA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PCSTR pszFormat,
  va_list         argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbVPrintfW and **RtlStringCbVPrintfA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string, along with a set of arguments in a **va_list**-typed argument list, and return a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbVPrintfW** and **RtlStringCbVPrintfA** to ensure that they do not write past the end of the buffer.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCbVPrintfW** to handle Unicode strings and **RtlStringCbVPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbVPrintfW**
char	"string"	**RtlStringCbVPrintfA**

If *pszDest* and *pszFormat*point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined..

Neither *pszFormat* nor *pszDest* should be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbVPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintf

RtlStringCbVPrintfEx

RtlStringCchVPrintf

RtlStringCbVPrintfExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbVPrintfExW** and **RtlStringCbVPrintfExA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbVPrintfExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cbDest,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcbRemaining,
  DWORD           dwFlags,
  NTSTRSAFE_PCSTR pszFormat,
  va_list         argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbVPrintfExW and **RtlStringCbVPrintfExA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string, along with a set of arguments in a **va_list**-typed argument list, and return a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbVPrintfExW** and **RtlStringCbVPrintfExA** to ensure that they do not write past the end of the buffer.

RtlStringCbVPrintfExW and **RtlStringCbVPrintfExA** add to the functionality of RtlStringCbVPrintf by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCbVPrintfExW** to handle Unicode strings and **RtlStringCbVPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbVPrintfExW**
char	"string"	**RtlStringCbVPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings, or if any argument strings overlap, behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszFormat* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCbVPrintf

RtlStringCchVPrintfEx

RtlStringCbVPrintfExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCbVPrintfExW** and **RtlStringCbVPrintfExA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbVPrintfExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcbRemaining,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

If *pszDest* is **NULL**, *cbDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcbRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbVPrintfExW and **RtlStringCbVPrintfExA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string, along with a set of arguments in a **va_list**-typed argument list, and return a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbVPrintfExW** and **RtlStringCbVPrintfExA** to ensure that they do not write past the end of the buffer.

RtlStringCbVPrintfExW and **RtlStringCbVPrintfExA** add to the functionality of RtlStringCbVPrintf by returning a pointer to the end of the destination string, as well as the number of bytes left unused in that string. Flags can be passed to the function for additional control.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCbVPrintfExW** to handle Unicode strings and **RtlStringCbVPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbVPrintfExW**
char	"string"	**RtlStringCbVPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings, or if any argument strings overlap, behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszFormat* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCbVPrintf

RtlStringCchVPrintfEx

RtlStringCbVPrintfW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCbVPrintfW** and **RtlStringCbVPrintfA** functions create a byte-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCbVPrintfW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cbDest,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*.

`cbDest`

The size of the destination buffer, in bytes. The buffer must be large enough to contain the formatted string plus the terminating null character.

For Unicode strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

For ANSI strings, the maximum number of bytes is NTSTRSAFE_MAX_CCH * sizeof(char).

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cbDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCbVPrintfW and **RtlStringCbVPrintfA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string, along with a set of arguments in a **va_list**-typed argument list, and return a formatted string. The size, in bytes, of the destination buffer is provided to **RtlStringCbVPrintfW** and **RtlStringCbVPrintfA** to ensure that they do not write past the end of the buffer.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCbVPrintfW** to handle Unicode strings and **RtlStringCbVPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCbVPrintfW**
char	"string"	**RtlStringCbVPrintfA**

If *pszDest* and *pszFormat*point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined..

Neither *pszFormat* nor *pszDest* should be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCbVPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintf

RtlStringCbVPrintfEx

RtlStringCchVPrintf

RtlStringCchCatA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCatW** and **RtlStringCchCatA** functions concatenate two character-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatW and **RtlStringCchCatA** should be used instead of the following functions:

- **strcat**
- **wcscat**

The size, in characters, of the destination buffer is provided to ensure that **RtlStringCchCatW** and **RtlStringCchCatA** do not write past the end of the buffer.

Use **RtlStringCchCatW** to handle Unicode strings and **RtlStringCchCatA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatW**
char	"string"	**RtlStringCchCatA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCatEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCchCatEx

RtlStringCchCatN

RtlStringCchCatExA function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCchCatExW** and **RtlStringCchCatExA** functions concatenate two character-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszSrc,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatExW and **RtlStringCchCatExA** should be used instead of the following functions:

- **strcat**
- **wcscat**

The size, in characters, of the destination buffer is provided to ensure that **RtlStringCchCatExW** and **RtlStringCchCatExA** do not write past the end of the buffer.

RtlStringCchCatExW and **RtlStringCchCatExA** add to the functionality of RtlStringCchCat by returning a pointer to the end of the destination string, as well as the number of characters left unused in the destination buffer. Flags can be passed to the function for additional control.

Use **RtlStringCchCatExW** to handle Unicode strings and **RtlStringCchCatExA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatExW**
char	"string"	**RtlStringCchCatExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCchCat

RtlStringCchCatNEx

RtlStringCchCatExW function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCchCatExW** and **RtlStringCchCatExA** functions concatenate two character-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszSrc,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcchRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatExW and **RtlStringCchCatExA** should be used instead of the following functions:

- **strcat**
- **wcscat**

The size, in characters, of the destination buffer is provided to ensure that **RtlStringCchCatExW** and **RtlStringCchCatExA** do not write past the end of the buffer.

RtlStringCchCatExW and **RtlStringCchCatExA** add to the functionality of RtlStringCchCat by returning a pointer to the end of the destination string, as well as the number of characters left unused in the destination buffer. Flags can be passed to the function for additional control.

Use **RtlStringCchCatExW** to handle Unicode strings and **RtlStringCchCatExA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatExW**
char	"string"	**RtlStringCchCatExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatEx

RtlStringCchCat

RtlStringCchCatNEx

RtlStringCchCatNA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCatNW** and **RtlStringCchCatNA** functions concatenate two character-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatNA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cchDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cchToAppend
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cchMaxAppend* characters, is added to the end of the string at *pszDest* and terminated with a null character.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

`cchToAppend`

The maximum number of characters to append to the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatNW and **RtlStringCchCatNA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in characters, of the destination buffer is provided to the function to ensure that **RtlStringCchCatNW** and **RtlStringCchCatNA** do not write past the end of the buffer.

Use **RtlStringCchCatNW** to handle Unicode strings and **RtlStringCchCatNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatNW**
char	"string"	**RtlStringCchCatNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatN

RtlStringCchCat

RtlStringCchCatNEx

RtlStringCchCatNExA function

1/25/2019 • 4 minutes to read • Edit Online

The RtlStringCchCatNExW and **RtlStringCchCatNExA** functions concatenate two character-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatNExA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cchDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cchToAppend,
  NTSTRSAFE_PSTR *ppszDestEnd,
  size_t         *pcchRemaining,
  DWORD          dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cchMaxAppend* characters, is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToAppend`

The maximum number of characters to append to the string that is contained in the buffer at *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatNExW and **RtlStringCchCatNExA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in characters, of the destination buffer is provided to **RtlStringCchCatNExW** and **RtlStringCchCatNExA** to ensure that the functions do not write past the end of the buffer.

RtlStringCchCatNExW and **RtlStringCchCatNExA** add to the functionality of RtlStringCchCatN by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchCatNExW** to handle Unicode strings and **RtlStringCchCatNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatNExW**
char	"string"	**RtlStringCchCatNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCchCatN

RtlStringCchCatNExW function

1/25/2019 • 4 minutes to read • Edit Online

The RtlStringCchCatNExW and **RtlStringCchCatNExA** functions concatenate two character-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatNExW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cchDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cchToAppend,
  NTSTRSAFE_PWSTR *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cchMaxAppend* characters, is added to the end of the string at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToAppend`

The maximum number of characters to append to the string that is contained in the buffer at *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the concatenation operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING
STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the concatenated string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatNExW and **RtlStringCchCatNExA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in characters, of the destination buffer is provided to **RtlStringCchCatNExW** and **RtlStringCchCatNExA** to ensure that the functions do not write past the end of the buffer.

RtlStringCchCatNExW and **RtlStringCchCatNExA** add to the functionality of RtlStringCchCatN by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchCatNExW** to handle Unicode strings and **RtlStringCchCatNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatNExW**
char	"string"	**RtlStringCchCatNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatNEx

RtlStringCchCatEx

RtlStringCchCatN

RtlStringCchCatNW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCatNW** and **RtlStringCchCatNA** functions concatenate two character-counted strings while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatNW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cchDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cchToAppend
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc*, up to *cchMaxAppend* characters, is added to the end of the string at *pszDest* and terminated with a null character.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

`cchToAppend`

The maximum number of characters to append to the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatNW and **RtlStringCchCatNA** should be used instead of the following functions:

- **strncat**
- **wcsncat**

The size, in characters, of the destination buffer is provided to the function to ensure that **RtlStringCchCatNW** and **RtlStringCchCatNA** do not write past the end of the buffer.

Use **RtlStringCchCatNW** to handle Unicode strings and **RtlStringCchCatNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatNW**
char	"string"	**RtlStringCchCatNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCatNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop

Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCatN

RtlStringCchCat

RtlStringCchCatNEx

RtlStringCchCatW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCatW** and **RtlStringCchCatA** functions concatenate two character-counted strings.

Syntax

```
NTSTRSAFEDDI RtlStringCchCatW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a buffer which, on input, contains a null-terminated string to which *pszSrc* will be concatenated. On output, this is the destination buffer that contains the entire resultant string. The string at *pszSrc* is added to the end of the string at *pszDest* and terminated with a null character.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer at *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCatW and **RtlStringCchCatA** should be used instead of the following functions:

- **strcat**
- **wcscat**

The size, in characters, of the destination buffer is provided to ensure that **RtlStringCchCatW** and **RtlStringCchCatA** do not write past the end of the buffer.

Use **RtlStringCchCatW** to handle Unicode strings and **RtlStringCchCatA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCatW**
char	"string"	**RtlStringCchCatA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCatEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCat

RtlStringCchCatEx

RtlStringCchCatN

RtlStringCchCopyA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCopyW** and **RtlStringCchCopyA** functions copy a null-terminated source string into a destination buffer of specified length.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • A **NULL** pointer was present. • The destination buffer length was zero at entry.

Remarks

RtlStringCchCopyW and **RtlStringCchCopyA** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

These functions are not replacements for **strncpy**. Use RtlStringCchCopyN or RtlStringCchCopyNEx to replace **strncpy**.

The size, in characters, of the destination buffer is provided to **RtlStringCchCopyW** and **RtlStringCchCopyA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCchCopyW** to handle Unicode strings and **RtlStringCchCopyA** to handle ANSI strings. The form you use depends your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyW**
char	"string"	**RtlStringCchCopyA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCopyEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCchCopyEx

RtlStringCchCopyExA function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCchCopyExW** and **RtlStringCchCopyExA** functions copy a character-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszSrc,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyExW and **RtlStringCchCopyExA** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

The size, in characters, of the destination buffer is provided to **RtlStringCchCopyExW** and **RtlStringCchCopyExA** to ensure that they do not write past the end of the buffer.

RtlStringCchCopyExW and **RtlStringCchCopyExA** add to the functionality of RtlStringCchCopy by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control. Use **RtlStringCchCopyExW** to handle Unicode strings and **RtlStringCchCopyExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyExW**
char	"string"	**RtlStringCchCopyExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyEx

RtlStringCchCopy

RtlStringCchCopyExW function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCchCopyExW** and **RtlStringCchCopyExA** functions copy a character-counted string into a buffer.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszSrc,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcchRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string. The *pszSrc* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyExW and **RtlStringCchCopyExA** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

The size, in characters, of the destination buffer is provided to **RtlStringCchCopyExW** and **RtlStringCchCopyExA** to ensure that they do not write past the end of the buffer.

RtlStringCchCopyExW and **RtlStringCchCopyExA** add to the functionality of RtlStringCchCopy by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control. Use **RtlStringCchCopyExW** to handle Unicode strings and **RtlStringCchCopyExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyExW**
char	"string"	**RtlStringCchCopyExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyEx

RtlStringCchCopy

RtlStringCchCopyNA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCopyNW** and **RtlStringCchCopyNA** functions copy a character-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyNA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cchDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cchToCopy
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc*, up to *cchSrc* characters, is copied to the buffer at *pszDest* and terminated with a null character.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cchToCopy`

The maximum number of characters to copy from *pszSrc* to the buffer that is supplied by *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyNW and **RtlStringCchCopyNA** should be used instead of **strncpy**.

The functions copy a given number of characters from a source string. **RtlStringCchCopyNW** and **RtlStringCchCopyNA** receive the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

Note that these functions behave differently from **strncpy** in one respect. If *cchSrc* is larger than the number of characters in *pszSrc*, **RtlStringCchCopyNW** and **RtlStringCchCopyNA**—unlike **strncpy**—do not continue to pad *pszDest* with null characters until *cchSrc* characters have been copied.

Use **RtlStringCchCopyNW** to handle Unicode strings and **RtlStringCchCopyNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyNW**
char	"string"	**RtlStringCchCopyNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCopyNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyN

RtlStringCchCopy

RtlStringCchCopyNEx

RtlStringCchCopyNExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA** functions copy a character-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyNExA(
  NTSTRSAFE_PSTR pszDest,
  size_t         cchDest,
  STRSAFE_PCNZCH pszSrc,
  size_t         cchToCopy,
  NTSTRSAFE_PSTR *ppszDestEnd,
  size_t         *pcchRemaining,
  DWORD          dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cchToCopy`

The maximum number of characters to copy from *pszSrc* to the buffer that is supplied by *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyNExW and **RtlStringCchCopyNExA** should be used instead of **strncpy**.

The functions copy a given number of characters from a source string. The size, in characters, of the destination buffer is provided to **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA** to ensure that they do not write past the end of the buffer.

Note that these functions behave differently from **strncpy** in one respect. If *cchSrc* is larger than the number of characters in *pszSrc*, **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA**—unlike **strncpy**—do not continue to pad *pszDest* with null characters until *cchSrc* characters have been copied.

RtlStringCchCopyNExW and **RtlStringCchCopyNExA** add to the functionality of RtlStringCchCopyN by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchCopyNExW** to handle Unicode strings and **RtlStringCchCopyNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyNExW**
char	"string"	**RtlStringCchCopyNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyNEx

RtlStringCchCopyN

RtlStringCchCopyNExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA** functions copy a character-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyNExW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cchDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cchToCopy,
  NTSTRSAFE_PWSTR *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cchToCopy`

The maximum number of characters to copy from *pszSrc* to the buffer that is supplied by *pszDest*.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyNExW and **RtlStringCchCopyNExA** should be used instead of **strncpy**.

The functions copy a given number of characters from a source string. The size, in characters, of the destination buffer is provided to **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA** to ensure that they do not write past the end of the buffer.

Note that these functions behave differently from **strncpy** in one respect. If *cchSrc* is larger than the number of characters in *pszSrc*, **RtlStringCchCopyNExW** and **RtlStringCchCopyNExA**—unlike **strncpy**—do not continue to pad *pszDest* with null characters until *cchSrc* characters have been copied.

RtlStringCchCopyNExW and **RtlStringCchCopyNExA** add to the functionality of RtlStringCchCopyN by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchCopyNExW** to handle Unicode strings and **RtlStringCchCopyNExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyNExW**
char	"string"	**RtlStringCchCopyNExA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyNEx

RtlStringCchCopyN

RtlStringCchCopyNW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCopyNW** and **RtlStringCchCopyNA** functions copy a character-counted string to a buffer while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyNW(
  NTSTRSAFE_PWSTR pszDest,
  size_t          cchDest,
  STRSAFE_PCNZWCH pszSrc,
  size_t          cchToCopy
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc*, up to *cchSrc* characters, is copied to the buffer at *pszDest* and terminated with a null character.

`cchDest`

The size of the destination buffer, in characters. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

`cchToCopy`

The maximum number of characters to copy from *pszSrc* to the buffer that is supplied by *pszDest*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchCopyNW and **RtlStringCchCopyNA** should be used instead of **strncpy**.

The functions copy a given number of characters from a source string. **RtlStringCchCopyNW** and **RtlStringCchCopyNA** receive the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

Note that these functions behave differently from **strncpy** in one respect. If *cchSrc* is larger than the number of characters in *pszSrc*, **RtlStringCchCopyNW** and **RtlStringCchCopyNA**—unlike **strncpy**—do not continue to pad *pszDest* with null characters until *cchSrc* characters have been copied.

Use **RtlStringCchCopyNW** to handle Unicode strings and **RtlStringCchCopyNA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyNW**
char	"string"	**RtlStringCchCopyNA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCopyNEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyN

RtlStringCchCopy

RtlStringCchCopyNEx

RtlStringCchCopyUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCopyUnicodeString** function copies the contents of a UNICODE_STRING structure to a specified destination.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyUnicodeString(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  PCUNICODE_STRING SourceString
);
```

Parameters

`pszDest`

A pointer to a buffer that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer at *pszDest* and terminated with a null character.

`cchDest`

The size, in characters, of the destination buffer. The buffer must be large enough for the string and the terminating null character. The maximum number of characters is NTSTRSAFE_MAX_CCH.

`SourceString`

A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

Return Value

RtlStringCchCopyUnicodeString returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlStringCchCopyUnicodeString returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The value in *cbDest* is larger than the maximum buffer size.
- The destination buffer (which *pszDest* points to) is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlStringCchCopyUnicodeString** function uses the destination buffer's size (which *cchDest* specifies) to ensure that the copy operation does not write past the end of the buffer.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString*and *pszDest* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlStringCchCopyUnicodeStringEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyUnicodeString

RtlStringCchCopyUnicodeStringEx

UNICODE_STRING

RtlStringCchCopyUnicodeStringEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlStringCchCopyUnicodeStringEx** function copies the contents of a UNICODE_STRING structure to a specified destination.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyUnicodeStringEx(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  PCUNICODE_STRING SourceString,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcchRemaining,
  DWORD            dwFlags
);
```

Parameters

`pszDest`

Optional. A pointer to a buffer that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer at *pszDest* and terminated with a null character. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size, in characters, of the destination buffer. The buffer must be large enough for the string and the terminating null character. The maximum number of characters in the buffer is NTSTRSAFE_MAX_CCH.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`ppszDestEnd`

Optional. If the caller supplies a non-**NULL** address pointer, then after the copy operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

Optional. If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters that are in the buffer that *pszDest* points to, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND_NULL

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.

STRSAFE_IGNORE_NULLS

If this flag is set, either the source or destination pointer, or both, can be **NULL**.
RtlStringCchCopyUnicodeStringEx treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

Return Value

RtlStringCchCopyUnicodeStringEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlStringCchCopyUnicodeStringEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The value in *cchDest* is larger than the maximum buffer size.
- The destination buffer (which *pszDest* points to) is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified.

- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL** or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlStringCchCopyUnicodeStringEx** function uses the destination buffer's size (which the *cchDest* parameter specifies) to ensure that the copy operation does not write past the end of the buffer.

RtlStringCchCopyUnicodeStringEx adds to the functionality of the RtlStringCchCopyUnicodeString function by returning a pointer to the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to the function for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString*and *pszDest* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these points can be **NULL**. If the *pszDest* pointer is **NULL**, the *SourceString*pointer must be **NULL** or the UNICODE_STRING structure must describe an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopyUnicodeStringEx

RtlStringCchCopyUnicodeString

UNICODE_STRING

RtlStringCchCopyW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchCopyW** and **RtlStringCchCopyA** functions copy a null-terminated source string into a destination buffer of specified length.

Syntax

```
NTSTRSAFEDDI RtlStringCchCopyW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives the copied string. The string at *pszSrc* is copied to the buffer at *pszDest* and terminated with a null character.

`cchDest`

The size, in characters, of the destination buffer. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszSrc`

A pointer to a caller-supplied, null-terminated string.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • A **NULL** pointer was present. • The destination buffer length was zero at entry.

Remarks

RtlStringCchCopyW and **RtlStringCchCopyA** should be used instead of the following functions:

- **strcpy**
- **wcscpy**

These functions are not replacements for **strncpy**. Use RtlStringCchCopyN or RtlStringCchCopyNEx to replace **strncpy**.

The size, in characters, of the destination buffer is provided to **RtlStringCchCopyW** and **RtlStringCchCopyA** to ensure that they do not write past the end of the buffer.

Use **RtlStringCchCopyW** to handle Unicode strings and **RtlStringCchCopyA** to handle ANSI strings. The form you use depends your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchCopyW**
char	"string"	**RtlStringCchCopyA**

If *pszSrc* and *pszDest* point to overlapping strings, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchCopyEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.

Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbCopy

RtlStringCchCopyEx

RtlStringCchLengthA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchLengthW** and **RtlStringCchLengthA** functions determine the length, in characters, of a supplied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchLengthA(
  STRSAFE_PCNZCH psz,
  size_t         cchMax,
  size_t         *pcchLength
);
```

Parameters

`psz`

A pointer to a buffer that contains a null-terminated string, the length of which will be checked.

`cchMax`

The maximum number of characters allowed in the buffer pointed to by *psz*, including the terminating null character. This value cannot exceed NTSTRSAFE_MAX_CCH.

`pcchLength`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in characters, of the string that is contained in the buffer. The length does not include the string's terminating null character.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	This success status means the string at *psz* was not **NULL**, and the length of the string (including the terminating null character) is less than or equal to *cchMax* characters.

STA TUS _IN VAL ID_ PAR AM ETE R	This error status means the value in *psz* is **NULL**, *cchMax* is larger than NTSTRSAFE_MAX_CCH, or *psz* is longer than *cchMax*.

Remarks

RtlStringCchLengthW and **RtlStringCchLengthA** should be used instead of **strlen**. They are used to ensure that a string is not larger than a given length, in characters. If that condition is met, **RtlStringCchLengthW** and **RtlStringCchLengthA** return the current length of the string in characters, not including the terminating null character.

Use **RtlStringCchLengthW** to handle Unicode strings and **RtlStringCchLengthA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchLengthW**
char	"string"	**RtlStringCchLengthA**

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbLength

RtlStringCchLengthW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchLengthW** and **RtlStringCchLengthA** functions determine the length, in characters, of a supplied string.

Syntax

```
NTSTRSAFEDDI RtlStringCchLengthW(
  STRSAFE_PCNZWCH psz,
  size_t          cchMax,
  size_t          *pcchLength
);
```

Parameters

`psz`

A pointer to a buffer that contains a null-terminated string, the length of which will be checked.

`cchMax`

The maximum number of characters allowed in the buffer pointed to by *psz*, including the terminating null character. This value cannot exceed NTSTRSAFE_MAX_CCH.

`pcchLength`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in characters, of the string that is contained in the buffer. The length does not include the string's terminating null character.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	This success status means the string at *psz* was not **NULL**, and the length of the string (including the terminating null character) is less than or equal to *cchMax* characters.

STA TUS _IN VAL ID_ PAR AM ETE R	This error status means the value in *psz* is **NULL**, *cchMax* is larger than NTSTRSAFE_MAX_CCH, or *psz* is longer than *cchMax*.

Remarks

RtlStringCchLengthW and **RtlStringCchLengthA** should be used instead of **strlen**. They are used to ensure that a string is not larger than a given length, in characters. If that condition is met, **RtlStringCchLengthW** and **RtlStringCchLengthA** return the current length of the string in characters, not including the terminating null character.

Use **RtlStringCchLengthW** to handle Unicode strings and **RtlStringCchLengthA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchLengthW**
char	"string"	**RtlStringCchLengthA**

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbLength

RtlStringCchPrintfA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchPrintfW** and **RtlStringCchPrintfA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchPrintfA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is **NTSTRSAFE_MAX_CCH**.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`arg4`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the output string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchPrintfW and **RtlStringCchPrintfA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments and return a formatted string. **RtlStringCchPrintfW** and **RtlStringCchPrintfA** accept the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

Use **RtlStringCchPrintfW** to handle Unicode strings and **RtlStringCchPrintfA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchPrintfW**
char	"string"	**RtlStringCchPrintfA**

If *pszDest*and *pszFormat* point to overlapping strings, or if any argument strings overlap, behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchPrintfEx.

Examples

The following example shows a simple use of **RtlStringCchPrintfW** using four arguments.

```
WCHAR pszDest[30];
size_t cchDest = 30;

LPCWSTR pszFormat = L"%s %d + %d = %d.";
WCHAR* pszTxt = L"The answer is";

NTSTATUS status =
    RtlStringCchPrintfW(pszDest, cchDest, pszFormat, pszTxt, 1, 2, 3);
```

The resultant string is "The answer is 1 + 2 = 3." It is contained in the buffer at *pszDest*. For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintf

RtlStringCchPrintfEx

RtlStringCchVPrintf

RtlStringCchPrintfExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchPrintfExW** and **RtlStringCchPrintfExA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchPrintfExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags,
  NTSTRSAFE_PCSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`arg7`

Return Value

The function returns one of the **NTSTATUS** values that are listed in the following table. For information about how to test **NTSTATUS** values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchPrintfExW and **RtlStringCchPrintfExA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments and return a formatted string. **RtlStringCchPrintfExW** and **RtlStringCchPrintfExA** accept the size, in characters, of the destination buffer to ensure that they do not write past the end of this buffer.

RtlStringCchPrintfExW and **RtlStringCchPrintfExA** add to the functionality of RtlStringCchPrintf by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchPrintfExW** to handle Unicode strings and **RtlStringCchPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchPrintfExW**
char	"string"	**RtlStringCchPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings, or any argument strings overlap, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCchPrintf

RtlStringCchVPrintfEx

RtlStringCchPrintfExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchPrintfExW** and **RtlStringCchPrintfExA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchPrintfExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcchRemaining,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting **NULL** string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused characters in the buffer pointed to by *pszDest*, including the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`arg7`

Return Value

The function returns one of the **NTSTATUS** values that are listed in the following table. For information about how to test **NTSTATUS** values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchPrintfExW and **RtlStringCchPrintfExA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments and return a formatted string. **RtlStringCchPrintfExW** and **RtlStringCchPrintfExA** accept the size, in characters, of the destination buffer to ensure that they do not write past the end of this buffer.

RtlStringCchPrintfExW and **RtlStringCchPrintfExA** add to the functionality of RtlStringCchPrintf by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

Use **RtlStringCchPrintfExW** to handle Unicode strings and **RtlStringCchPrintfExA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchPrintfExW**
char	"string"	**RtlStringCchPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings, or any argument strings overlap, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintfEx

RtlStringCchPrintf

RtlStringCchVPrintfEx

RtlStringCchPrintfW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchPrintfW** and **RtlStringCchPrintfA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchPrintfW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the function's argument list.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is **NTSTRSAFE_MAX_CCH**.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`arg4`

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the output string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer's length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchPrintfW and **RtlStringCchPrintfA** should be used instead of the following functions:

- **sprintf**
- **swprintf**
- **_snprintf**
- **_snwprintf**

All of these functions accept a format string and a list of arguments and return a formatted string. **RtlStringCchPrintfW** and **RtlStringCchPrintfA** accept the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

Use **RtlStringCchPrintfW** to handle Unicode strings and **RtlStringCchPrintfA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchPrintfW**
char	"string"	**RtlStringCchPrintfA**

If *pszDest*and *pszFormat* point to overlapping strings, or if any argument strings overlap, behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchPrintfEx.

Examples

The following example shows a simple use of **RtlStringCchPrintfW** using four arguments.

```
WCHAR pszDest[30];
size_t cchDest = 30;

LPCWSTR pszFormat = L"%s %d + %d = %d.";
WCHAR* pszTxt = L"The answer is";

NTSTATUS status =
    RtlStringCchPrintfW(pszDest, cchDest, pszFormat, pszTxt, 1, 2, 3);
```

The resultant string is "The answer is 1 + 2 = 3." It is contained in the buffer at *pszDest*. For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbPrintf

RtlStringCchPrintfEx

RtlStringCchVPrintf

RtlStringCchVPrintfA function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchVPrintfW** and **RtlStringCchVPrintfA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchVPrintfA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PCSTR pszFormat,
  va_list         argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchVPrintfW and **RtlStringCchVPrintfA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string and its arguments, which are provided as a **va_list**-typed argument list, and return a formatted string. **RtlStringCchVPrintfW** and **RtlStringCchVPrintfA** receive the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCchVPrintfW** to handle Unicode strings and **RtlStringCchVPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchVPrintfW**
char	"string"	**RtlStringCchVPrintfA**

If *pszDest* and *pszFormat* point to overlapping strings, or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchVPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintf

RtlStringCchPrintf

RtlStringCchVPrintfEx

RtlStringCchVPrintfExA function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchVPrintfExW** and **RtlStringCchVPrintfExA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchVPrintfExA(
  NTSTRSAFE_PSTR  pszDest,
  size_t          cchDest,
  NTSTRSAFE_PSTR  *ppszDestEnd,
  size_t          *pcchRemaining,
  DWORD           dwFlags,
  NTSTRSAFE_PCSTR pszFormat,
  va_list         argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including those bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows.

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

Pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchVPrintfExW and **RtlStringCchVPrintfExA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string and its arguments, provided as a **va_list**-typed argument list, and return a formatted string. **RtlStringCchVPrintfExW** and **RtlStringCchVPrintfExA** receive the size, in characters, of the destination buffer to ensure that they do not write past the end of the buffer.

RtlStringCchVPrintfExW and **RtlStringCchVPrintfExA** add to the functionality of RtlStringCchVPrintf by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCchVPrintfExW** to handle Unicode strings and **RtlStringCchVPrintfExA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchVPrintfExW**
char	"string"	**RtlStringCchVPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintfEx

RtlStringCchPrintfEx

RtlStringCchVPrintf

RtlStringCchVPrintfExW function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlStringCchVPrintfExW** and **RtlStringCchVPrintfExA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchVPrintfExW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PWSTR  *ppszDestEnd,
  size_t           *pcchRemaining,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*. The *pszDest* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH. If *pszDest* is **NULL**, *cchDest* must be zero.

`ppszDestEnd`

If the caller supplies a non-**NULL** address pointer then, after the operation completes, the function loads that address with a pointer to the destination buffer's resulting null string terminator.

`pcchRemaining`

If the caller supplies a non-**NULL** address pointer, the function loads the address with the number of unused bytes that are in the buffer pointed to by *pszDest*, including those bytes used for the terminating null character.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows.

VALUE	MEANING

STRSAFE_FILL_BEHIND_NULL	If set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the terminating null character.
STRSAFE_IGNORE_NULLS	If set, either *pszDest*or*pszSrc*, or both, can be **NULL**. **NULL** *pszSrc* pointers are treated like empty strings (TEXT("")), which can be copied. **NULL** *pszDest* pointers cannot receive nonempty strings.
STRSAFE_FILL_ON_FAILURE	If set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer, and the buffer is null-terminated. This operation overwrites any preexisting buffer contents.
STRSAFE_NULL_ON_FAILURE	If set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.
STRSAFE_NO_TRUNCATION	If set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

`pszFormat`

Pointer to a null-terminated text string that contains **printf**-styled formatting directives. The *pszFormat* pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • An invalid flag was specified. • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present without the STRSAFE_IGNORE_NULLS flag. • The destination buffer pointer was **NULL**, but the buffer size was not zero. • The destination buffer pointer was **NULL**, or its length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchVPrintfExW and **RtlStringCchVPrintfExA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string and its arguments, provided as a **va_list**-typed argument list, and return a formatted string. **RtlStringCchVPrintfExW** and **RtlStringCchVPrintfExA** receive the size, in characters, of the destination buffer to ensure that they do not write past the end of the buffer.

RtlStringCchVPrintfExW and **RtlStringCchVPrintfExA** add to the functionality of RtlStringCchVPrintf by returning a pointer to the end of the destination string, as well as the number of characters left unused in that string. Flags can be passed to the function for additional control.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCchVPrintfExW** to handle Unicode strings and **RtlStringCchVPrintfExA** to handle ANSI strings. The form you use depends on your data.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchVPrintfExW**
char	"string"	**RtlStringCchVPrintfExA**

If *pszDest* and *pszFormat* point to overlapping strings or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszSrc* nor *pszDest* can be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set, in which case either or both can be **NULL**. If *pszDest* is **NULL**, *pszSrc* must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintfEx

RtlStringCchPrintfEx

RtlStringCchVPrintf

RtlStringCchVPrintfW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringCchVPrintfW** and **RtlStringCchVPrintfA** functions create a character-counted text string, with formatting that is based on supplied formatting information.

Syntax

```
NTSTRSAFEDDI RtlStringCchVPrintfW(
  NTSTRSAFE_PWSTR  pszDest,
  size_t           cchDest,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`pszDest`

A pointer to a caller-supplied buffer that receives a formatted, null-terminated string. The function creates this string from both the formatting string that is supplied by *pszFormat* and the arguments supplied by *argList*.

`cchDest`

The size of the destination buffer, in characters. The buffer must be large enough to contain the formatted string plus the terminating null character. The maximum number of characters allowed is NTSTRSAFE_MAX_CCH.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`argList`

A **va_list**-typed argument list. Arguments contained in the argument list will be interpreted by using the formatting string that is supplied by *pszFormat*.

Return Value

The function returns one of the NTSTATUS values that are listed in the following table. For information about how to test NTSTATUS values, see Using NTSTATUS Values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the output string was created without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means the operation did not complete due to insufficient space in the destination buffer. The destination buffer contains a truncated version of the created string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph. The function returns the STATUS_INVALID_PARAMETER value when: • The value in *cchDest* is larger than the maximum buffer size. • The destination buffer was already full. • A **NULL** pointer was present. • The destination buffer length was zero, but a nonzero length source string was present.

Remarks

RtlStringCchVPrintfW and **RtlStringCchVPrintfA** should be used instead of the following functions:

- **vsprintf**
- **vswprintf**
- **_vsnprintf**
- **_vsnwprintf**

All of these functions accept a format string and its arguments, which are provided as a **va_list**-typed argument list, and return a formatted string. **RtlStringCchVPrintfW** and **RtlStringCchVPrintfA** receive the size, in characters, of the destination buffer to ensure that the functions do not write past the end of the buffer.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

Use **RtlStringCchVPrintfW** to handle Unicode strings and **RtlStringCchVPrintfA** to handle ANSI strings. The form you use depends on your data, as shown in the following table.

STRING DATA TYPE	STRING LITERAL	FUNCTION
WCHAR	L"string"	**RtlStringCchVPrintfW**
char	"string"	**RtlStringCchVPrintfA**

If *pszDest* and *pszFormat* point to overlapping strings, or if any argument strings overlap, the behavior of the function is undefined.

Neither *pszFormat* nor *pszDest* can be **NULL**. If you need to handle **NULL** string pointer values, use RtlStringCchVPrintfEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbVPrintf

RtlStringCchPrintf

RtlStringCchVPrintfEx

RtlUnalignedStringCbLengthW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnalignedStringCbLengthW** function is a version of the RtlStringCbLength function that accepts an unaligned pointer to a string of Unicode characters.

Syntax

```
NTSTRSAFEDDI RtlUnalignedStringCbLengthW(
  STRSAFE_PCUNZWCH psz,
  size_t           cbMax,
  size_t           *pcbLength
);
```

Parameters

`psz`

Supplies a pointer to a buffer that contains a null-terminated string whose length **RtlUnalignedStringCbLengthW** will check.

`cbMax`

Supplies the maximum number of bytes that are allowed in the buffer that *psz* points to, including the terminating NULL character. This value cannot exceed NTSTRSAFE_MAX_CCH * sizeof(WCHAR).

`pcbLength`

Optional. If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in bytes, of the string that is contained in the buffer that *psz* points to. The length does not include the string's terminating NULL character.

Return Value

RtlUnalignedStringCbLengthW returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	This *success* status means the string that the *psz* parameter specified was not **NULL**, and the length of the string (including the terminating NULL character) was less than or equal to *cbMax* bytes.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means the value in *psz* is **NULL**, *cbMax* is larger than NTSTRSAFE_MAX_CCH * sizeof(WCHAR, or *psz* is longer than *cbMax*.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnalignedStringCbLengthW** function is available for processor architectures, such as Itanium-based and x64-based, that cause alignment exceptions when software attempts to access unaligned data. On those processors, you can use **RtlUnalignedStringCbLengthW** instead of the RtlStringCbLength function to avoid alignment exceptions. (For processors that do not cause alignment exceptions, **RtlUnalignedStringCbLengthW** is equivalent to **RtlStringCbLength**.)

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCbLength

RtlUnalignedStringCchLengthW function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnalignedStringCchLengthW** function is a version of the RtlStringCchLength function that accepts an unaligned pointer to a string of Unicode characters.

Syntax

```
NTSTRSAFEDDI RtlUnalignedStringCchLengthW(
  STRSAFE_PCUNZWCH psz,
  size_t           cchMax,
  size_t           *pcchLength
);
```

Parameters

`psz`

Supplies a pointer to a buffer that contains a null-terminated string whose length **RtlUnalignedStringCchLengthW** will check.

`cchMax`

Supplies the maximum number of characters that are allowed in the buffer that *psz* points to, including the terminating NULL character. This value cannot exceed NTSTRSAFE_MAX_CCH.

`pcchLength`

Optional. If the caller supplies a non-**NULL** address pointer, the function loads the address with the length, in characters, of the string that is contained in the buffer that *psz* points to. The length does not include the string's terminating NULL character.

Return Value

RtlUnalignedStringCchLengthW returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means the string that the *psz* parameter was not **NULL**, and the length of the string (including the terminating NULL character) was less than or equal to *cchMax* characters.

STATUS_INVALID_PARAMETER	This *error* status means the value in *psz* is **NULL**, *cchMax* is larger than NTSTRSAFE_MAX_CCH, or *psz* is longer than *cchMax*.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnalignedStringCchLengthW** function is available for processor architectures, such as Itanium-based and x64-based, that cause alignment exceptions when software attempts to access unaligned data. On those processors, you can use **RtlUnalignedStringCchLengthW** instead of RtlStringCchLength to avoid alignment exceptions. (For processors that do not cause alignment exceptions, **RtlUnalignedStringCchLengthW** is equivalent to **RtlStringCchLength**.)

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlStringCchLength

RtlUnicodeStringCat function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCat** function concatenates two strings that are contained in UNICODE_STRING structures.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCat(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`SourceString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

Return Value

RtlUnicodeStringCat returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCat returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCat** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use RtlUnicodeStringCatEx.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCatEx

UNICODE_STRING

RtlUnicodeStringCatEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCatEx** function concatenates two strings that are contained in UNICODE_STRING structures.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCatEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString*can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCatEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCatEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means the concatenation operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCatEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCatEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCatEx adds to the functionality of the RtlUnicodeStringCat function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCatEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must describe an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCat

UNICODE_STRING

RtlUnicodeStringCatString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCatString** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCatString(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating null) is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`pszSrc`

A pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the buffer that the **UNICODE_STRING** structure points to.

Return Value

RtlUnicodeStringCatString returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCatString returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCatString** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCatStringEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCatStringEx

UNICODE_STRING

RtlUnicodeStringCatStringEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCatStringEx** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCatStringEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating null) is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

Optional. A caller-supplied pointer to a null-terminated source string. This string will be concatenated to the end of the string that is contained in the **UNICODE_STRING** structure that *DestinationString* specifies. *pszSrc* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCatStringEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCatStringEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCatStringEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCatStringEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCatStringEx adds to the functionality of the RtlUnicodeStringCatString function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCatStringEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCatString

UNICODE_STRING

RtlUnicodeStringCbCatN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCbCatN** function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCatN(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cbToAppend
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`SourceString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`cbToAppend`

The maximum number of bytes to append to the string that the *DestinationString* parameter describes.

Return Value

RtlUnicodeStringCbCatN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCatN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cbToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCatN** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCbCatNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)

Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatNEx

RtlUnicodeStringCchCatN

UNICODE_STRING

RtlUnicodeStringCbCatNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCbCatNEx** function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCatNEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cbToAppend,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToAppend`

The maximum number of bytes to append to the string that the *DestinationString* parameter describes.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCbCatNEx** treats

NULL source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCbCatNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCatNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in dwFlags.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cbToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCatNEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCbCatNEx adds to the functionality of the RtlUnicodeStringCbCatN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCbCatNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must specify an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatN

RtlUnicodeStringCchCatNEx

UNICODE_STRING

RtlUnicodeStringCbCatStringN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCbCatStringN** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCatStringN(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cbToAppend
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating null) is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`pszSrc`

A caller-supplied pointer to a null-terminated string. This string will be concatenated to the end of the destination string that is contained in the **UNICODE_STRING** structure that *DestinationString* points to.

`cbToAppend`

The maximum number of bytes to append to the string that the *DestinationString* parameter describes.

Return Value

RtlUnicodeStringCbCatStringN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCatStringN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cbToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCatStringN** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCbCatStringNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)

Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatStringNEx

RtlUnicodeStringCchCatStringN

UNICODE_STRING

RtlUnicodeStringCbCatStringNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCbCatStringNEx** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCatStringNEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cbToAppend,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating null) is added to the end of the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

A caller-supplied pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the **UNICODE_STRING** structure that *DestinationString* points to. *pszSrc* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToAppend`

The maximum number of bytes to append to the string that the *DestinationString* parameter describes.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCbCatStringNEx**

treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCbCatStringNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenated operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCatStringNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cbToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCatStringNEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCbCatStringNEx adds to the functionality of the RtlUnicodeStringCbCatStringN function by returning a **UNICODE_STRING** structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCbCatStringNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatStringN

RtlUnicodeStringCchCatStringNEx

UNICODE_STRING

RtlUnicodeStringCbCopyN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCbCopyN** function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCopyN(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cbToCopy
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the *DestinationString*structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`SourceString`

A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`cbToCopy`

The number of bytes to be copied from the source to the destination.

Return Value

RtlUnicodeStringCbCopyN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation.

STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph.

RtlUnicodeStringCbCopyN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cbToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCopyN** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCbCopyNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)

Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyNEx

RtlUnicodeStringCchCopyN

UNICODE_STRING

RtlUnicodeStringCbCopyNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCbCopyNEx** function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCopyNEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cbToCopy,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the *DestinationString* structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).*DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToCopy`

The number of bytes to be copied from the source to the destination.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCbCopyNEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers

cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCbCopyNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCopyNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in dwFlags.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cbToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCopyNEx** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCbCopyNEx adds to the functionality of the RtlUnicodeStringCbCopyN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCbCopyNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must contain an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyN

RtlUnicodeStringCchCopyNEx

UNICODE_STRING

RtlUnicodeStringCbCopyStringN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCbCopyStringN** function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCopyStringN(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cbToCopy
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc* parameter points to (excluding the terminating null) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`pszSrc`

A pointer to the string to be copied.

`cbToCopy`

The number of bytes to be copied from the source to the destination.

Return Value

RtlUnicodeStringCbCopyN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCopyN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cbToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCopyN** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCbCopyNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)

Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyNEx

RtlUnicodeStringCchCopyN

UNICODE_STRING

RtlUnicodeStringCbCopyStringNEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeStringCbCopyStringNEx** function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCbCopyStringNEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cbToCopy,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc* parameter points to (excluding the terminating null) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

Optional. A pointer to the string to be copied. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cbToCopy`

The number of bytes to be copied from the source to the destination.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCbCopyStringNEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCbCopyStringNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCbCopyStringNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.

- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cbToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCbCopyStringNEx** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCbCopyStringNEx adds to the functionality of the RtlUnicodeStringCbCopyStringN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes left unused in that string. You can pass flags **RtlUnicodeStringCbCopyStringNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyStringN

RtlUnicodeStringCchCopyStringNEx

UNICODE_STRING

RtlUnicodeStringCchCatN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCchCatN** function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCatN(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cchToAppend
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a destination string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`SourceString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`cchToAppend`

The maximum number of characters to append to the string that the *DestinationString* parameter describes.

Return Value

RtlUnicodeStringCchCatN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STA TUS _BU FFE R_O VER FLO W	This *warning* status means that the concatenated operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCatN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cchToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCatN** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCchCatNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatN

RtlUnicodeStringCchCatNEx

UNICODE_STRING

RtlUnicodeStringCchCatNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCchCatNEx** function concatenates two strings that are contained in UNICODE_STRING structures while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCatNEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cchToAppend,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string is added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToAppend`

The maximum number of characters to append to the string that the *DestinationString* parameter describes.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCchCatNEx** treats

NULL source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCchCatNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenated operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCatNEx returns the STATUS_INVALID_PARAMETER value when:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cchToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCatNEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCchCatNEx adds to the functionality of the RtlUnicodeStringCchCatN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCchCatNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must specify an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatNEx

RtlUnicodeStringCchCatN

UNICODE_STRING

RtlUnicodeStringCchCatStringN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCchCatStringN** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCatStringN(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cchToAppend
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating **NULL**) is added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`pszSrc`

A caller-supplied pointer to a null-terminated string. This string will be concatenated to the end of the string that is contained in the *DestinationString* parameter's **UNICODE_STRING** structure.

`cchToAppend`

The maximum number of characters to append to the string that the *DestinationString* parameter describes.

Return Value

RtlUnicodeStringCchCatStringN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the strings were concatenated without truncation, and the resultant destination buffer is null-terminated.

STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenated operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCatStringN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cchToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCatStringN** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character value (that is, with zero).

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCchCatStringNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)

Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatStringN

RtlUnicodeStringCchCatStringNEx

UNICODE_STRING

RtlUnicodeStringCchCatStringNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCchCatStringNEx** function concatenates two strings when the destination string is contained in a UNICODE_STRING structure, while limiting the size of the appended string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCatStringNEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cchToAppend,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that, on input, contains a string to which the source string will be concatenated. On output, this buffer is the destination buffer that contains the entire resultant string. The source string (excluding the terminating null) is added to the end of the destination string. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

A caller-supplied pointer to a null-terminated string. This string will be concatenated to the end of the string that *DestinationString* describes. *pszSrc* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToAppend`

The maximum number of characters to append to the string that the *DestinationString* parameter describes.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCchCatStringNEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer

pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCchCatStringNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the concatenation operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the intended result.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCatStringNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cchToAppend* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCatStringNEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCchCatStringNEx adds to the functionality of the RtlUnicodeStringCchCatStringN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. Flags can be passed to **RtlUnicodeStringCchCatStringNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCatStringNEx

RtlUnicodeStringCchCatStringN

UNICODE_STRING

RtlUnicodeStringCchCopyN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCchCopyN** function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCopyN(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cchToCopy
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`SourceString`

A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`cchToCopy`

The number of characters to be copied from the source to the destination.

Return Value

RtlUnicodeStringCchCopyN returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STA TUS _BU FFE R_O VER FLO W	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.
STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCopyN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cchToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCopyN** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCchCopyNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCchCopyN

RtlUnicodeStringCchCopyNEx

UNICODE_STRING

RtlUnicodeStringCchCopyNEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringCchCopyNEx** function copies a string from one UNICODE_STRING structure to another while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCopyNEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  size_t           cchToCopy,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *SourceString* parameter's **UNICODE_STRING** structure points to is copied to the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure that contains the string to be copied. The maximum number of characters in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToCopy`

The number of characters to be copied from the source to the destination.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCchCopyNEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers

cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCchCopyNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCopyNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.

- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cchToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCopyNEx** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCchCopyNEx adds to the functionality of the RtlUnicodeStringCchCopyN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCchCopyNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must contain an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyNEx

RtlUnicodeStringCchCopyN

UNICODE_STRING

RtlUnicodeStringCchCopyStringN function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCchCopyStringN** function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCopyStringN(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cchToCopy
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc*parameter points to (excluding the terminating **NULL**) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`pszSrc`

A pointer to the string to be copied.

`cchToCopy`

The number of characters to be copied from the source to the destination.

Return Value

RtlUnicodeStringCchCopyStringN returns one of the following NTSTATUS.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCopyStringN returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.
- The *cchToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCopyN** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc*and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCchCopyNEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCchCopyN

RtlUnicodeStringCchCopyNEx

UNICODE_STRING

RtlUnicodeStringCchCopyStringNEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeStringCchCopyStringNEx** function copies a string into a UNICODE_STRING structure while limiting the size of the copied string.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCchCopyStringNEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  size_t           cchToCopy,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc* parameter points to (excluding the terminating null) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

Optional. A pointer to the string to be copied. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`cchToCopy`

The number of characters to be copied from the source to the destination.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCchCopyStringNEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCchCopyStringNEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCchCopyStringNEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.

- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.
- The *cchToCopy* parameter's value is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCchCopyStringNEx** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCchCopyStringNEx adds to the functionality of the RtlUnicodeStringCchCopyStringN function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCchCopyStringNEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCbCopyStringNEx

RtlUnicodeStringCchCopyStringN

UNICODE_STRING

RtlUnicodeStringCopy function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCopy** function copies a string from one UNICODE_STRING structure to another.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCopy(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure. The source string is copied to the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

`SourceString`

A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be copied to the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).

Return Value

RtlUnicodeStringCopy returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated version of the source string.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCopy returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCopy** function uses the destination buffer's size to ensure that the copy operation does not write past the end of the buffer. If the string must be truncated at the end of the destination buffer, the function does not terminate the resultant string with a null character.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCopyEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCopyEx

UNICODE_STRING

RtlUnicodeStringCopyEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeStringCopyEx** function copies a string from one UNICODE_STRING structure to another.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCopyEx(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure. The source string is copied to the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure. This structure includes a buffer that contains the source string. This string will be copied to the destination string. The maximum number of bytes in the structure's string buffer is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *SourceString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the copied string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCopyEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCopyEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCopyEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCopyEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCopyEx adds to the functionality of the RtlUnicodeStringCopy function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCopyEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *SourceString* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *SourceString* pointer must be **NULL** or the **UNICODE_STRING** structure must describe an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCopy

UNICODE_STRING

RtlUnicodeStringCopyString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringCopyString** function copies a string into a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCopyString(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc*parameter points to (excluding the terminating null) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * **sizeof**(WCHAR).

`pszSrc`

A pointer to a null-terminated string. This string will be copied into the buffer that the *DestinationString*parameter's **UNICODE_STRING** structure points to.

Return Value

RtlUnicodeStringCopyString returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCopyString returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure are invalid.
- The destination buffer is already full.
- A buffer pointer is **NULL**.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCopyString** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringCopyStringEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCopyStringEx

UNICODE_STRING

RtlUnicodeStringCopyStringEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeStringCopyStringEx** function copies a string into a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringCopyStringEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives the copied string. The string that the *pszSrc* parameter points to (excluding the terminating null) is copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. The maximum number of bytes in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR). *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

Optional. A pointer to a null-terminated string that will be copied into the buffer that the *DestinationString* parameter's **UNICODE_STRING** structure points to. *pszSrc* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, the function sets this structure's **Buffer** member to the end of the concatenated string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringCopyStringEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

Return Value

RtlUnicodeStringCopyStringEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the copy operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringCopyStringEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringCopyStringEx** function uses the destination buffer's size to ensure that the concatenation operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringCopyStringEx adds to the functionality of the RtlUnicodeStringCopyString function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringCopyStringEx** for additional control.

If the source and destination strings overlap, the behavior of the function is undefined.

The *pszSrc* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszSrc* pointer must be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringCopyString

UNICODE_STRING

RtlUnicodeStringInit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringInit** function initializes a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringInit(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure to be initialized. The *pszSrc*pointer is copied into the *DestinationString* parameter's **UNICODE_STRING** structure. The maximum number of characters in the string that *pszSrc* points to is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`pszSrc`

Optional. A pointer to a null-terminated string constant. This string pointer will be copied to the **Buffer** member of the **UNICODE_STRING** structure pointed to by the *DestinationString* parameter. This string pointer can be **NULL**.

Return Value

RtlUnicodeStringInit returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringInit returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The length of the string that the *pszSrc* parameter points to is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH characters.

- The *DestinationString* pointer is **NULL**, but the *pszSrc* pointer is not **NULL**.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringInit** function does the following:

- Sets the **Length** member of the UNICODE_STRING structure that is pointed to by the *pszSrc* parameter to the length, in bytes, of the string that the *pszSrc* parameter points to.
- Sets the **MaximumLength** member of the **UNICODE_STRING** structure to the length, in bytes, of the string that the *pszSrc* parameter points to plus 2 bytes.
- Sets the **Buffer** member of the **UNICODE_STRING** structure to the address that the *pszSrc* parameter specifies.

The *DestinationString* pointer should not be **NULL**. If you need to specify **NULL** *DestinationString* pointer values, use the RtlUnicodeStringInitEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringInitEx

UNICODE_STRING

RtlUnicodeStringInitEx function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringInitEx** function initializes a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringInitEx(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszSrc,
  DWORD            dwFlags
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure to be initialized. The *pszSrc* pointer is copied into the *DestinationString* parameter's **UNICODE_STRING** structure. The maximum number of characters in the string that *pszSrc* points to is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`pszSrc`

Optional. A pointer to a null-terminated string constant. This string pointer will be copied to the **Buffer** member of the **UNICODE_STRING** structure pointed to by the *DestinationString* parameter. This string pointer can be **NULL**.

`dwFlags`

The following flag is defined:

STRSAFE_IGNORE_NULLS

If this flag is set, the source pointer can be **NULL**. **RtlUnicodeStringInitEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied.

Return Value

RtlUnicodeStringInitEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STA **TUS** **_SU** **CCE** **SS**	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.

STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringInitEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The length of the string that the *pszSrc* parameter points to is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH characters.
- The *DestinationString* pointer is **NULL**, but the *pszSrc* pointer is not **NULL**.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringInitEx** function does the following:

- Sets the **Length** member of the UNICODE_STRING structure to the length, in bytes, of the string that the *pszSrc* parameter points to.
- Sets the **MaximumLength** member of the **UNICODE_STRING** structure to the length, in bytes, of the string that the *pszSrc* parameter points to plus 2 bytes.
- Sets the **Buffer** member of the **UNICODE_STRING** structure to the address that the *pszSrc* parameter specifies.

The *DestinationString* pointer cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringInit

UNICODE_STRING

RtlUnicodeStringPrintf function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringPrintf** function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringPrintf(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives a formatted string. **RtlUnicodeStringPrintf** creates this string from the formatting string that *pszFormat* specifies and the function's argument list. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`arg3`

Return Value

RtlUnicodeStringPrintf returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the operation did not complete because of insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringPrintf returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure that the *DestinationString* parameter points to are invalid.
- The destination buffer is already full.
- A **NULL** pointer is present.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringPrintf** function uses the destination buffer's size to ensure that the string formatting operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character.

If the format string and destination string overlap, the behavior of the function is undefined.

The *pszFormat* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringPrintfEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringPrintfEx

RtlUnicodeStringVPrintf

RtlUnicodeStringVPrintfEx

RtlUnicodeStringPrintfEx function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeStringPrintfEx** function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringPrintfEx(
  PUNICODE_STRING  DestinationString,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  ...
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives a formatted string. **RtlUnicodeStringPrintfEx** creates this string from both the formatting string that *pszFormat* specifies and the function's argument list. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, **RtlUnicodeStringPrintfEx** sets this structure's **Buffer** member to the end of the formatted string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringPrintfEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation

overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`arg5`

Return Value

RtlUnicodeStringPrintfEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means that the operation did not complete because of insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringPrintfEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.
- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.

- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringPrintfEx** function uses the destination buffer's size to ensure that the string formatting operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringPrintfEx adds to the functionality of the RtlUnicodeStringPrintf function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringPrintfEx** for additional control.

If the format string and the destination string overlap, the behavior of the function is undefined.

The *pszFormat* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszFormat* pointer must either be **NULL** or point to an empty string.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringPrintf

RtlUnicodeStringVPrintf

RtlUnicodeStringVPrintfEx

UNICODE_STRING

RtlUnicodeStringValidate function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringValidate** function validates the contents of a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringValidate(
  PCUNICODE_STRING SourceString
);
```

Parameters

`SourceString`

A pointer to a **UNICODE_STRING** structure to be validated.

Return Value

RtlUnicodeStringValidate returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that the function completed successfully.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

RtlUnicodeStringValidate returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- *SourceString*->**Length** % sizeof(WCHAR) does not equal zero.
- *SourceString*->**MaximumLength** % sizeof(WCHAR) does not equal zero.
- *SourceString*->**Length** is greater than *SourceString*->**MaximumLength**.
- *SourceString*->**MaximumLength** is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).
- *SourceString*->**Buffer** equals **NULL** but *SourceString*->**Length** or *SourceString*->**MaximumLength** does not equal zero.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The *SourceString* pointer cannot be **NULL**. If you need to specify a **NULL** pointer value, use the RtlUnicodeStringValidateEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringValidateEx

UNICODE_STRING

RtlUnicodeStringValidateEx function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringValidateEx** function validates the contents of a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringValidateEx(
  PCUNICODE_STRING SourceString,
  DWORD            dwFlags
);
```

Parameters

`SourceString`

Optional. A pointer to a **UNICODE_STRING** structure to be validated. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

The following flag is defined:

STRSAFE_IGNORE_NULLS

If this flag is set, the source pointer can be **NULL**. **RtlUnicodeStringValidateEx** treats **NULL** source buffer pointers like empty strings (TEXT("")).

Return Value

RtlUnicodeStringValidateEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means that the function completed successfully.
STATUS_INVALID_PARAMETER	This *error* status means that the function received an invalid input parameter. For more information, see the following list.

If STRSAFE_IGNORE_NULLS is not set in dwFlags, **RtlUnicodeStringValidateEx** returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- *SourceString*->**Length** % sizeof(WCHAR) does not equal zero.
- *SourceString*->**MaximumLength** % sizeof(WCHAR) does not equal zero.
- *SourceString*->**Length** is greater than *SourceString*->**MaximumLength**.
- *SourceString*->**MaximumLength** is greater than NTSTRSAFE_UNICODE_STRING_MAX_CCH * sizeof(WCHAR).
- *SourceString*->**Buffer** equals **NULL** but *SourceString*->Length or *SourceString*->**MaximumLength** does not equal zero.
- An invalid flag is specified for *dwFlags*.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The *SourceString* pointer cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringValidate

UNICODE_STRING

RtlUnicodeStringVPrintf function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringVPrintf** function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringVPrintf(
  PUNICODE_STRING  DestinationString,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`DestinationString`

A pointer to a **UNICODE_STRING** structure that receives a formatted string. **RtlUnicodeStringVPrintf** creates this string from the formatting string that *pszFormat* specifies and the function's argument list. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH.

`pszFormat`

A pointer to a null-terminated text string that contains **printf**-styled formatting directives.

`argList`

A **va_list**-typed argument list. Arguments in this argument list will be interpreted by using the formatting string that *pszFormat* specifies.

Return Value

RtlUnicodeStringVPrintf returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, the string was copied without truncation, and the resultant destination buffer is null-terminated.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient buffer space. The destination buffer contains a truncated, null-terminated version of the intended result.

STA TUS _IN VAL ID_ PAR AM ETE R	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph.

RtlUnicodeStringVPrintf returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of the **UNICODE_STRING** structure that *DestinationString* points to are invalid.
- The destination buffer is already full.
- A **NULL** pointer is present.
- The destination buffer's length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringVPrintf** function uses the destination buffer's size to ensure that the string formatting operation does not write past the end of the buffer. The function does not terminate the resultant string with a null character.

If the format string and destination string overlap, the behavior of the function is undefined.

The *pszFormat* and *DestinationString* pointers cannot be **NULL**. If you need to handle **NULL** pointer values, use the RtlUnicodeStringVPrintfEx function.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available in Windows XP with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib

IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringPrintf

RtlUnicodeStringPrintfEx

RtlUnicodeStringVPrintfEx

UNICODE_STRING

RtlUnicodeStringVPrintfEx function

1/25/2019 • 4 minutes to read • Edit Online

The **RtlUnicodeStringVPrintfEx** function creates a text string, with formatting that is based on supplied formatting information, and stores the string in a UNICODE_STRING structure.

Syntax

```
NTSTRSAFEDDI RtlUnicodeStringVPrintfEx(
  PUNICODE_STRING  DestinationString,
  PUNICODE_STRING  RemainingString,
  DWORD            dwFlags,
  NTSTRSAFE_PCWSTR pszFormat,
  va_list          argList
);
```

Parameters

`DestinationString`

Optional. A pointer to a **UNICODE_STRING** structure that receives a formatted string. **RtlUnicodeStringVPrintfEx** creates this string from the formatting string that *pszFormat* supplies and the function's argument list. The maximum number of characters in the string is NTSTRSAFE_UNICODE_STRING_MAX_CCH. *DestinationString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`RemainingString`

Optional. If the caller supplies a non-**NULL** pointer to a **UNICODE_STRING** structure, **RtlUnicodeStringVPrintfE** sets this structure's **Buffer** member to the end of the formatted string, sets the structure's **Length** member to zero, and sets the structure's **MaximumLength** member to the number of bytes that are remaining in the destination buffer. *RemainingString* can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`dwFlags`

One or more flags and, optionally, a fill byte. The flags are defined as follows:

STRSAFE_FILL_BEHIND

If this flag is set and the function succeeds, the low byte of *dwFlags* is used to fill the portion of the destination buffer that follows the last character in the string.

STRSAFE_IGNORE_NULLS

If this flag is set, the source or destination pointer, or both, can be **NULL**. **RtlUnicodeStringVPrintfEx** treats **NULL** source buffer pointers like empty strings (TEXT("")), which can be copied. **NULL** destination buffer pointers cannot receive nonempty strings.

STRSAFE_FILL_ON_FAILURE

If this flag is set and the function fails, the low byte of *dwFlags* is used to fill the entire destination buffer. This operation overwrites any preexisting buffer contents.

STRSAFE_NULL_ON_FAILURE

If this flag is set and the function fails, the destination buffer is set to an empty string (TEXT("")). This operation

overwrites any preexisting buffer contents.

STRSAFE_NO_TRUNCATION

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the contents of the destination buffer are not modified.

STRSAFE_ZERO_LENGTH_ON_FAILURE

If this flag is set and the function returns STATUS_BUFFER_OVERFLOW, the destination string length is set to zero bytes.

`pszFormat`

A pointer to a nul-terminated text string that contains **printf**-styled formatting directives. This pointer can be **NULL**, but only if STRSAFE_IGNORE_NULLS is set in *dwFlags*.

`argList`

A **va_list**-typed argument list. Arguments in this argument list will be interpreted by the using formatting string that *pszFormat* supplies.

Return Value

RtlUnicodeStringVPrintfEx returns one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	This *success* status means source data was present, and the strings were concatenated without truncation.
STATUS_BUFFER_OVERFLOW	This *warning* status means the copy operation did not complete due to insufficient space in the destination buffer. If STRSAFE_NO_TRUNCATION is set in *dwFlags*, the destination buffer is not modified. If the flag is not set, the destination buffer contains a truncated version of the copied string.
STATUS_INVALID_PARAMETER	This *error* status means the function received an invalid input parameter. For more information, see the following paragraph.

RtlUnicodeStringVPrintfEx returns the STATUS_INVALID_PARAMETER value when one of the following occurs:

- The contents of a **UNICODE_STRING** structure are invalid.
- An invalid flag is specified in *dwFlags*.

- The destination buffer is already full.
- A buffer pointer is **NULL** and the STRSAFE_IGNORE_NULLS flag is not specified in *dwFlags*.
- The destination buffer pointer is **NULL**, but the buffer size is not zero.
- The destination buffer pointer is **NULL**, or its length is zero, but a nonzero length source string is present.

For information about how to test NTSTATUS values, see Using NTSTATUS Values.

Remarks

The **RtlUnicodeStringVPrintfEx** function uses the destination buffer's size to ensure that the string formatting operation does not write past the end of the buffer. By default, the function does not terminate the resultant string with a null character value (that is, with zero). As an option, the caller can use the STRSAFE_FILL_BEHIND flag and a fill byte value of zero to null-terminate a resultant string that does not occupy the entire destination buffer.

RtlUnicodeStringVPrintfEx adds to the functionality of the RtlUnicodeStringVPrintf function by returning a UNICODE_STRING structure that identifies the end of the destination string and the number of bytes that are left unused in that string. You can pass flags to **RtlUnicodeStringVPrintfEx** for additional control.

If the format string and the destination string overlap, the behavior of the function is undefined.

The *pszFormat* and *DestinationString* pointers cannot be **NULL** unless the STRSAFE_IGNORE_NULLS flag is set in *dwFlags*. If STRSAFE_IGNORE_NULLS is set, one or both of these pointers can be **NULL**. If the *DestinationString* pointer is **NULL**, the *pszFormat* pointer must be **NULL** or point to an empty string.

For more information about **va_list**-typed argument lists, see the Microsoft Windows SDK documentation.

For more information about the safe string functions, see Using Safe String Functions.

Requirements

Minimum supported client	Available starting with Windows XP with Service Pack 1 (SP1).
Target Platform	Desktop
Header	ntstrsafe.h (include Ntstrsafe.h)
Library	Ntstrsafe.lib
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringPrintf

RtlUnicodeStringPrintfEx

RtlUnicodeStringVPrintf

UNICODE_STRING

pcivirt.h header

1/8/2019 • 8 minutes to read • Edit Online

Reference guide for using interfaces used exposing VFs to a Virtual Machine.

Devices that conform to the PCI Express Single-Root I/O Virtualization (SR-IOV) specification can provide multiple interfaces to the device. These interfaces, known as Virtual Functions (VFs), are independent and are provided through the initial device interface, known as the Physical Function (PF). For example, an Ethernet NIC that supports SR-IOV can be designed to have a switch with one physical Ethernet port (connected to the physical wire) and many virtual Ethernet ports.

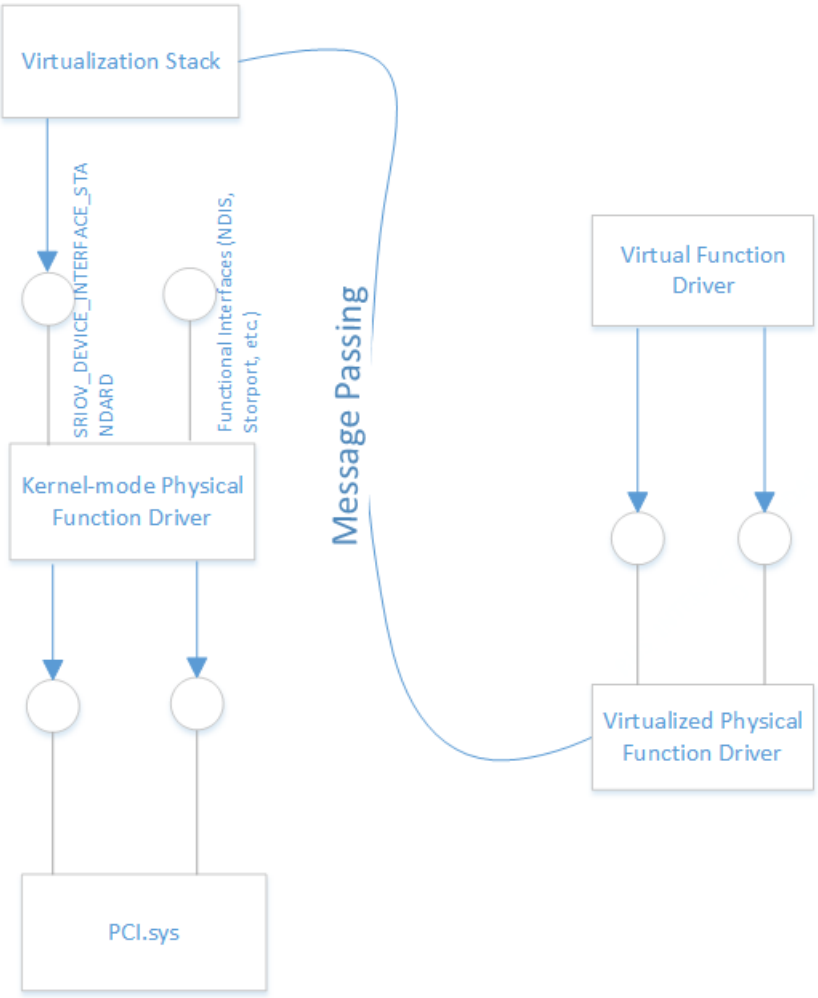
The PF's configuration space allows the PF driver to manage the VF's PCI resources, including memory-mapped I/O space and message-signaled interrupts. Since VFs are a subset of a complete device, they can be less expensive to expose in hardware than a traditional function in a multi-function package. This allows the device maker to build more interfaces, and to manage any shared resources centrally.

When Windows is running directly on the machine hardware, the device drivers participate in operations related to Plug and Play, power management, interrupt management and other tasks. A trusted Windows bus driver and Hardware Abstraction Layer (HAL) own bus configuration and configure the entire bus. The driver runs within the same privilege level and there are no trust boundaries in kernel mode.

When Windows is running on a virtual machine (VM), those assumptions do not apply. VFs can be put under the control of a non-privileged VM. However, the hardware must be security checked so that the security or performance of the system is not impacted.

When a driver running on the VF requests a configuration space read or write, the request is received by the virtualization stack and sent to the SR-IOV device's PF driver. It is the responsibility of the PF driver to respond to those requests and provide details for the VF. The PF driver may occasionally require that a configuration read or write request that is passed down to the hardware.

The stack uses an I/O MMU to differentiate traffic coming from the various interfaces that the device exposes, enforcing policy about which regions of memory a device can access and which interrupts it can generate.



Hardware requirements

The system to be used for SR-IOV device assignment must meet the requirements for SR-IOV networking and Direct Device Assignment. The system must have a IOMMU, that IOMMU must be configured to give control of devices over to the operating system, and PCIe ACS (Access Control Services) must be enabled and configured for use by the operating system. Finally, the device in question must not use line based interrupts, and must not require ATS (Address Translation Services).

More information here:

Everything you wanted to know about SR-IOV in Hyper-V. Part 1

Discrete Device Assignment — Description and background

To determine if a system supports device assignement, and if a particular PCI device will work for device assignment:

Discrete Device Assignment script

Querying for SR-IOV devices

GUID_DEVINTERFACE_VIRTUALIZABLE_DEVICE is a device class interface that is provided by drivers for SR-IOV devices. This GUID provides a way to query for all the device stacks which expose the various function tables that are used to manage the virtualization-related features of the device. After the driver registers the GUID, individual capabilities are discovered by sending IRP_MN_QUERY_INTERFACE. The driver must respond to that request with GUID_SRIOV_DEVICE_INTERFACE_STANDARD. Drivers must also handle IOCTL_SRIOV_NOTIFICATION and IOCTL_SRIOV_EVENT_COMPLETE.

A driver for an SR_IOV device, which runs in a privileged VM is the host OS. It owns Plug-and-Play and power

management for a entire machine, and exposes PCI Express SR-IOV Virtual Functions in non-privileged VMs, must provide the GUID_SRIOV_DEVICE_INTERFACE_STANDARD (defined in the header Pcivirt.h). That driver might be PCI Express SR-IOV Physical Function (PF) driver that creates the FDO, or it might be a lower filter on that device node in the case when the FDO is being managed by a port driver.

The device interface is required so that the driver can access the configuration space of the VFs.

In the PF driver's EVT_WDF_DRIVER_DEVICE_ADD implementation, perform these tasks:

- After calling WdfDeviceCreate to create the function device object (FDO), call WdfDeviceCreateDeviceInterface to register GUID_DEVINTERFACE_VIRTUALIZABLE_DEVICE. This allows the virtualization stack to retrieve a device handle to the SR-IOV device.
- Expose the GUID_SRIOV_DEVICE_INTERFACE_STANDARD.
    - Initialize a SRIOV_DEVICE_INTERFACE_STANDARD structure and set members to function pointers of the callback functions implemented by the PF driver.
    - Configure the structure by calling WDF_QUERY_INTERFACE_CONFIG_INIT.
    - Register the interface with the FDO by calling WdfDeviceAddQueryInterface.

```
    // Make the device visible as an assignable device.
    //
    status = WdfDeviceCreateDeviceInterface(
        fdo,
        &GUID_DEVINTERFACE_VIRTUALIZABLE_DEVICE,
        NULL);
    if (!NT_SUCCESS(status))
    {
        TraceEvents(TRACE_LEVEL_ERROR, DBG_INIT,
                    "Failed to create interface: %!STATUS!",
                    status);
        goto Cleanup;
    }

    //
    // Expose SRIOV_DEVICE_INTERFACE_STANDARD
    //
    RtlZeroMemory(&sriovInterface, sizeof(sriovInterface));
    sriovInterface.Size = sizeof(sriovInterface);
    sriovInterface.Version = 1;
    sriovInterface.Context = deviceContext;
    sriovInterface.InterfaceReference = Virtualization_ReferenceInterface;
    sriovInterface.InterfaceDereference = Virtualization_DereferenceInterface;
    sriovInterface.ReadVfConfig = Virtualization_ReadConfig;
    sriovInterface.WriteVfConfig = Virtualization_WriteConfig;
    sriovInterface.ReadVfConfigBlock = Virtualization_ReadBlock;
    sriovInterface.WriteVfConfigBlock = Virtualization_WriteBlock;
    sriovInterface.ResetVf = Virtualization_ResetFunction;
    sriovInterface.SetVfPowerState = Virtualization_SetPowerState;
    sriovInterface.GetDeviceLocation = Virtualization_GetDeviceLocation;
    sriovInterface.GetVendorAndDevice = Virtualization_GetVendorAndDevice;
    sriovInterface.QueryProbedBars = Virtualization_QueryProbedBars;
    sriovInterface.QueryLuid = Virtualization_QueryLuid;


    WDF_QUERY_INTERFACE_CONFIG_INIT(&qiConfig,
                                    (PINTERFACE)&sriovInterface,
                                    &GUID_SRIOV_DEVICE_INTERFACE_STANDARD,
                                    NULL);

    status = WdfDeviceAddQueryInterface(fdo, &qiConfig);

    if (!NT_SUCCESS(status))
    {
        TraceEvents(TRACE_LEVEL_ERROR, DBG_INIT,
                    "WdfDeviceAddQueryInterface failed: %!STATUS!\n",
                    status);
        goto Cleanup;
    }
```

Handling Plug and Play events

The virtualization stack is responsible for sending the appropriate messages to the VM, waiting for the reply (with a timeout) and in case of non-responding VM, and applying appropriate action such as vetoing the PnP event, or surprise-removing the device from the non-privileged VM. PF drivers that implement GUID_DEVINTERFACE_VIRTUALIZABLE_DEVICE must also handle these I/O control requests that allow the virtualization stack to react to PnP events.

- The virtualization stack first sends IOCTL_SRIOV_ATTACH to the device. This notifies the device that the virtualization stack needs to be notified about certain PnP events.
- This is in effect until the virtualization stack sends IOCTL_SRIOV_DETACH.
- The virtualization stack queries the devices about PnP events by sending IOCTL_SRIOV_NOTIFICATION

requests. The PF driver can inform the virtualization stack of a PnP event by completing the IOCTL_SRIOV_NOTIFICATION request.

- The virtualization stack unblocks those events by sending IOCTL_SRIOV_EVENT_COMPLETE. pcivirt.h contains the following programming interfaces:

IOCTLs

TITLE	DESCRIPTION
IOCTL_SRIOV_ATTACH	The request indicates that the virtualization stack wants to register for Plug and Play events received by the SR-IOV device.
IOCTL_SRIOV_DETACH	The request indicates that the virtualization stack wants to unregister for Plug and Play events (previously registered through the IOCTL_SRIOV_ATTACH request).
IOCTL_SRIOV_EVENT_COMPLETE	The request indicates that the virtualization stack or the SR-IOV device received one of the events listed in SRIOV_PF_EVENT.
IOCTL_SRIOV_INVALIDATE_BLOCK	The IOCTL_SRIOV_INVALIDATE_BLOCK request indicates that the virtualization stack wants to reset the contents of the specified configuration block.
IOCTL_SRIOV_MITIGATED_RANGE_UPDATE	The IOCTL_SRIOV_MITIGATED_RANGE_UPDATE request indicates that the virtualization stack wants to update to the mitigation ranges.
IOCTL_SRIOV_NOTIFICATION	The request indicates that the virtualization stack wants to be notified when one of the events listed in SRIOV_PF_EVENT occurs.
IOCTL_SRIOV_PROXY_QUERY_LUID	This request supplies the local unique identifier of the SR_IOV device implementing the interface.
IOCTL_SRIOV_QUERY_MITIGATED_RANGE_COUNT	The request determines the ranges of memory-mapped I/O space that must mitigated.
IOCTL_SRIOV_QUERY_MITIGATED_RANGES	The request determines the specific ranges on which intercepts must be placed.

Callback functions

TITLE	DESCRIPTION
READ_WRITE_MITIGATED_REGISTER	Reads or writes to mitigated address spaces.
SRIOV_GET_DEVICE_LOCATION	Retrieves information about the current location of the PCI device on the bus, such as PCI Segment, Bus, Device and Function number.
SRIOV_GET_MMIO_REQUIREMENTS	This callback function is not supported.

TITLE	DESCRIPTION
SRIOV_GET_RESOURCE_FOR_BAR	Gets the translated resource for a specific Base Address Register (BAR).
SRIOV_GET_VENDOR_AND_DEVICE_IDS	Supplies the Vendor and Device ID for a PCI Express SR-IOV Virtual Function (VF) to be used for generating a more generic Plug and Play ID for the VF. These IDs cannot be read directly from the VF's configuration space.
SRIOV_QUERY_LUID	Gets the local unique identifier of the SR-IOV device.
SRIOV_QUERY_LUID_VF	Gets the PCI Express SR-IOV Virtual Function (VF) given a unique identifier.
SRIOV_QUERY_PROBED_BARS	Queries the data read from the physical function's (PF) base address registers (BARs) if the value -1 were written to them first.
SRIOV_QUERY_PROBED_BARS_2	Queries the data read from the specified PCI Express SR-IOV Virtual Function (VF) base address registers (BARs) if the value -1 were written to them first.
SRIOV_QUERY_VF_LUID	Gets the local unique identifier of the PCI Express SR-IOV Virtual Function (VF).
SRIOV_READ_BLOCK	Reads data from the specified configuration block of a PCI Express SR-IOV Virtual Function (VF).
SRIOV_READ_CONFIG	Reads data from the configuration space of the specified PCI Express SR-IOV Virtual Function (VF).
SRIOV_RESET_FUNCTION	Resets the specified PCI Express SR-IOV Virtual Function (VF).
SRIOV_SET_POWER_STATE	Sets the power state of the specified PCI Express SR-IOV Virtual Function (VF).
SRIOV_WRITE_BLOCK	Writes data to the specified configuration block of a PCI Express SR-IOV Virtual Function (VF).
SRIOV_WRITE_CONFIG	Writes configuration data to a PCI Express SR-IOV Virtual Function (VF).

Structures

TITLE	DESCRIPTION
MITIGABLE_DEVICE_INTERFACE	Stores function pointers to callback functions implemented by the physical function (PF) driver for the mitigable device interface.
SRIOV_DEVICE_INTERFACE_STANDARD	Stores function pointers to callback functions implemented by the physical function (PF) driver in the device stack for the of the SR-IOV device.

TITLE	DESCRIPTION
SRIOV_DEVICE_INTERFACE_STANDARD_2	Stores function pointers to callback functions implemented by the physical function (PF) driver in the device stack for the of the SR-IOV device. This is an extended version of SRIOV_DEVICE_INTERFACE_STANDARD.
SRIOV_INVALIDATE_BLOCK	Contains the configuration block information. This structure is used in a IOCTL_SRIOV_INVALIDATE_BLOCK request.
SRIOV_MITIGATED_RANGE_COUNT_INPUT	This structure is used as an input buffer to the IOCTL_SRIOV_QUERY_MITIGATED_RANGE_COUNT request to determine the ranges of memory-mapped I/O space that must be mitigated.
SRIOV_MITIGATED_RANGE_COUNT_OUTPUT	This structures is the output buffer received by the IOCTL_SRIOV_QUERY_MITIGATED_RANGE_COUNT request that contains an array of ranges of memory-mapped I/O space that must be mitigated.
SRIOV_MITIGATED_RANGE_UPDATE_INPUT	This structure is used as an input buffer to the IOCTL_SRIOV_MITIGATED_RANGE_UPDATE request to indicate the virtual function (VF) whose memory-mapped I/O space that must be mitigated.
SRIOV_MITIGATED_RANGE_UPDATE_OUTPUT	This structures is the output buffer received by the IOCTL_SRIOV_MITIGATED_RANGE_UPDATE request that indicates the virtual function (VF) whose memory-mapped I/O space was mitigated.
SRIOV_MITIGATED_RANGES_INPUT	This structure is the input buffer in the IOCTL_SRIOV_QUERY_MITIGATED_RANGES request to get the specific ranges on which intercepts must be placed.
SRIOV_MITIGATED_RANGES_OUTPUT	This structure is the output buffer received by the IOCTL_SRIOV_QUERY_MITIGATED_RANGES request to get the specific ranges on which intercepts must be placed.
SRIOV_PNP_EVENT_COMPLETE	Stores the status for an event that the SR-IOV Physical Function (PF) driver should set for Plug and Play even completion. This structure is used in the input buffer of the IOCTL_SRIOV_EVENT_COMPLETE request.
SRIOV_PROXY_QUERY_LUID_OUTPUT	Stores the local unique identifier of the SR_IOV device implementing the interface. This structure is the output buffer for the IOCTL_SRIOV_PROXY_QUERY_LUID request.

Enumerations

TITLE	DESCRIPTION
SRIOV_PF_EVENT	Defines event values for the SR-IOV device.

pep_x.h header

1/8/2019 • 15 minutes to read • Edit Online

This header contains programming interfaces used by the Power Engine Plugins (PEPs). For more information, see:

- Platform Extension Plug-ins (PEPs) pep_x.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE	The PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE	The PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

TITLE	DESCRIPTION
PoFxRegisterCoreDevice	The PoFxRegisterCoreDevice routine registers a new core system resource with the Windows power management framework (PoFx).
PoFxRegisterPlugin	The PoFxRegisterPlugin routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPluginEx	The PoFxRegisterPluginEx routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Callback functions

TITLE	DESCRIPTION
PEPCALLBACKNOTIFYACPI	An AcceptAcpiNotification event callback routine handles ACPI notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYDPM	An AcceptDeviceNotification event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYPPM	An AcceptProcessorNotification event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKPOWERONCRASHDUMPDEVICE	The PowerOnDumpDeviceCallback callback routine turns on the crash-dump device.
POFXCALLBACKCRITICALRESOURCE	The TransitionCriticalResource routine changes the active/inactive state of a core system component.
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS	The EnumerateUnmaskedInterrupts routine enumerates interrupt sources whose interrupts are unmasked and enabled.
POFXCALLBACKPLATFORMIDLEVETO	The PlatformIdleVeto routine increments or decrements the veto count for a veto code for a platform idle state.
POFXCALLBACKPROCESSORHALT	The ProcessorHalt routine prepares the processor to be halted.
POFXCALLBACKPROCESSORIDLEVETO	The ProcessorIdleVeto routine increments or decrements the veto count for a veto code for a processor idle state.
POFXCALLBACKREQUESTCOMMON	The RequestCommon routine is a generic request handler.
POFXCALLBACKREQUESTINTERRUPT	The RequestInterrupt routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.

TITLE	DESCRIPTION
POFXCALLBACKREQUESTWORKER	The RequestWorker routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.
POFXCALLBACKUPDATEPLATFORMIDLESTATE	The UpdatePlatformIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.
POFXCALLBACKUPDATEPROCESSORIDLESTATE	The UpdateProcessorIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK	An EnumerateInterruptSource callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.

Structures

TITLE	DESCRIPTION
PEP_ABANDON_DEVICE	The PEP_ABANDON_DEVICE structure identifies a device that has been abandoned and will no longer be used by the operating system.
PEP_ACPI_ABANDON_DEVICE	The PEP_ACPI_ABANDON_DEVICE structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE	The PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure contains an enumeration of the objects in the namespace of the device.
PEP_ACPI_EVALUATE_CONTROL_METHOD	The PEP_ACPI_EVALUATE_CONTROL_METHOD structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.
PEP_ACPI_EXTENDED_ADDRESS	The PEP_ACPI_EXTENDED_ADDRESS structure is used to report resource usage in the address space such as memory and IO.
PEP_ACPI_GPIO_RESOURCE	The PEP_ACPI_GPIO_RESOURCE structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.
PEP_ACPI_INTERRUPT_RESOURCE	The PEP_ACPI_INTERRUPT_RESOURCE structure describes an ACPI interrupt resource.
PEP_ACPI_IO_MEMORY_RESOURCE	The PEP_ACPI_IO_MEMORY_RESOURCE structure describes an ACPI IO port descriptor resource.
PEP_ACPI_OBJECT_NAME	The PEP_ACPI_OBJECT_NAME union contains the four-character name of an ACPI object.

TITLE	DESCRIPTION
PEP_ACPI_OBJECT_NAME_WITH_TYPE	The PEP_ACPI_OBJECT_NAME_WITH_TYPE structure that specifies both the path-relative name of an ACPI object and the type of this object.
PEP_ACPI_PREPARE_DEVICE	The PEP_ACPI_PREPARE_DEVICE structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure contains a list of raw resources that are needed to control power to the device.
PEP_ACPI_QUERY_OBJECT_INFORMATION	The PEP_ACPI_QUERY_OBJECT_INFORMATION structure contains information about an ACPI object.
PEP_ACPI_REGISTER_DEVICE	The PEP_ACPI_REGISTER_DEVICE structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.
PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES	The PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.
PEP_ACPI_RESOURCE	The PEP_ACPI_RESOURCE structure contains hardware details for a specific ACPI resource.
PEP_ACPI_RESOURCE_FLAGS	The PEP_ACPI_RESOURCE_FLAGS structure contains flags describing an ACPI resource.
PEP_ACPI_SPB_I2C_RESOURCE	The PEP_ACPI_SPB_I2C_RESOURCE structure describes an ACPI I2C serial bus resource.
PEP_ACPI_SPB_RESOURCE	The PEP_ACPI_SPB_RESOURCE structure describes an ACPI serial bus connection resource.
PEP_ACPI_SPB_SPI_RESOURCE	The PEP_ACPI_SPB_SPI_RESOURCE structure describes an ACPI SPI serial bus resource.
PEP_ACPI_SPB_UART_RESOURCE	The PEP_ACPI_SPB_UART_RESOURCE structure describes an ACPI UART serial bus resource.
PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.
PEP_ACPI_UNREGISTER_DEVICE	The PEP_ACPI_UNREGISTER_DEVICE structure contains information about a device that has been unregistered from ACPI services.
PEP_COMPONENT_ACTIVE	The PEP_COMPONENT_ACTIVE structure identifies a component that is making a transition between the idle condition and the active condition.
PEP_COMPONENT_PERF_INFO	The PEP_COMPONENT_PERF_INFO structure describes the performance states (P-states) of a component.

TITLE	DESCRIPTION
PEP_COMPONENT_PERF_SET	The PEP_COMPONENT_PERF_SET structure describes the performance states (P-states) in a P-state set.
PEP_COMPONENT_PERF_STATE_REQUEST	The PEP_COMPONENT_PERF_STATE_REQUEST structure specifies a performance state (P-state) set and a new performance level to assign to this set.
PEP_COMPONENT_PLATFORM_CONSTRAINTS	The PEP_COMPONENT_PLATFORM_CONSTRAINTS structure describes the lowest-powered Fx state of that a component can be in when the platform is in a particular idle state.
PEP_COMPONENT_V2	The PEP_COMPONENT_V2 structure specifies the power state attributes of a component in the device.
PEP_COORDINATED_DEPENDENCY_OPTION	The PEP_COORIDNATED_DEPENDENCY_OPTION structure describes a coordinated idle state's dependency to the OS.
PEP_COORDINATED_IDLE_STATE	The PEP_COORIDNATED_IDLE_STATE structure describes a coordinated idle state to the OS.
PEP_CRASHDUMP_INFORMATION	The PEP_CRASHDUMP_INFORMATION structure contains information about a crash-dump device.
PEP_DEBUGGER_TRANSITION_REQUIREMENTS	The PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure indicates the platform idle states for which the debugger device must be turned on.
PEP_DEVICE_PLATFORM_CONSTRAINTS	The PEP_DEVICE_PLATFORM_CONSTRAINTS structure specifies the constraints for entry to the various Dx power states that are supported by a device.
PEP_DEVICE_POWER_STATE	The PEP_DEVICE_POWER_STATE structure indicates the status of a transition to a new Dx (device power) state.
PEP_DEVICE_REGISTER_V2	The PEP_DEVICE_REGISTER structure describes all the components in a particular device.
PEP_DEVICE_STARTED	The PEP_DEVICE_STARTED structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).
PEP_INFORMATION	The PEP_INFORMATION structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V1	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).

TITLE	DESCRIPTION
PEP_KERNEL_INFORMATION_STRUCT_V2	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V3	The PEP_KERNEL_INFORMATION structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_LOW_POWER_EPOCH	The PEP_LOW_POWER_EPOCH structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).
PEP_NOTIFY_COMPONENT_IDLE_STATE	The PEP_NOTIFY_COMPONENT_IDLE_STATE structure contains status information about a component's pending transition to a new Fx power state.
PEP_PERF_STATE	The PEP_PERF_STATE structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.
PEP_PLATFORM_IDLE_STATE	The PEP_PLATFORM_IDLE_STATE structure specifies the properties of a platform idle state.
PEP_PLATFORM_IDLE_STATE_UPDATE	The PEP_PLATFORM_IDLE_STATE_UPDATE structure contains the updated properties of a platform idle state.
PEP_POWER_CONTROL_COMPLETE	The PEP_POWER_CONTROL_COMPLETE structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.
PEP_POWER_CONTROL_REQUEST	The PEP_POWER_CONTROL_REQUEST structure contains a request from a driver for a power control operation.
PEP_PPM_CONTEXT_QUERY_PARKING_PAGE	The PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure describes the parking page for a processor.
PEP_PPM_CST_STATE	The PEP_PPM_CST_STATE structure specifies the properties of a C state (ACPI processor power state).
PEP_PPM_CST_STATES	The PEP_PPM_CST_STATES structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.
PEP_PPM_ENTER_SYSTEM_STATE	Used in the PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE notification to notify PEP that the system is about to enter a system power state. .
PEP_PPM_FEEDBACK_READ	The PEP_PPM_FEEDBACK_READ structure contains the value read from a processor performance feedback counter.
PEP_PPM_IDLE_CANCEL	The PEP_PPM_IDLE_CANCEL structure indicates why the processor could not enter the previously selected idle state.

TITLE	DESCRIPTION
PEP_PPM_IDLE_COMPLETE	The PEP_PPM_IDLE_COMPLETE structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_COMPLETE_V2	The PEP_PPM_IDLE_COMPLETE_V2 structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_EXECUTE	The PEP_PPM_IDLE_EXECUTE structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_EXECUTE_V2	The PEP_PPM_IDLE_EXECUTE_V2 structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_SELECT	The PEP_PPM_IDLE_SELECT structure describes the most energy-efficient idle state that the processor can enter and still satisfy the constraints specified by the operating system.
PEP_PPM_INITIATE_WAKE	The PEP_PPM_INITIATE_WAKE structure indicates whether a processor requires an interrupt to wake up from an idle state.
PEP_PPM_IS_PROCESSOR_HALTED	The PEP_PPM_IS_PROCESSOR_HALTED structure indicates whether the processor is currently halted in its selected idle state.
PEP_PPM_LPI_COMPLETE	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_PARK_MASK	The PEP_PROCESSOR_PARK_MASK structure contains the current core parking mask.
PEP_PPM_PARK_SELECTION	The PEP_PPM_PARK_SELECTION structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PARK_SELECTION_V2	The PEP_PPM_PARK_SELECTION_V2 structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PERF_CHECK_COMPLETE	The PEP_PPM_PERF_CHECK_COMPLETE structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.
PEP_PPM_PERF_CONSTRAINTS	The PEP_PPM_PERF_CONSTRAINTS structure describes the performance limits to apply to the processor.
PEP_PPM_PERF_SET	The PEP_PPM_PERF_SET structure specifies the new performance level that the operating system is requesting for the processor.

TITLE	DESCRIPTION
PEP_PPM_PERF_SET_STATE	Used in the PEP_NOTIFY_PPM_PERF_SET notification at runtime to set the current operating performance of the processor. .
PEP_PPM_PLATFORM_STATE_RESIDENCIES	The PEP_PPM_PLATFORM_STATE_RESIDENCIES structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.
PEP_PPM_PLATFORM_STATE_RESIDENCY	The PEP_PPM_PLATFORM_STATE_RESIDENCY structure specifies the accumulated residency time and transition count for a particular platform idle state.
PEP_PPM_QUERY_CAPABILITIES	The PEP_PPM_QUERY_CAPABILITIES structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).
PEP_PPM_QUERY_COORDINATED_DEPENDENCY	The PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure describes dependencies for coordinated idle states.
PEP_PPM_QUERY_COORDINATED_STATES	The PEP_PPM_QUERY_COORDINATED_STATES structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.
PEP_PPM_QUERY_DISCRETE_PERF_STATES	Used in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification that stores the list of discrete performance states that PEP supports, if the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification indicates support for discrete performance states. .
PEP_PPM_QUERY_DOMAIN_INFO	Used in the PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO notification that queries for information about a performance domain. .
PEP_PPM_QUERY_FEEDBACK_COUNTERS	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_QUERY_IDLE_STATES	The PEP_PPM_QUERY_IDLE_STATES structure describes the idle states of a particular processor.
PEP_PPM_QUERY_IDLE_STATES_V2	The PEP_PPM_QUERY_IDLE_STATES_V2 structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.
PEP_PPM_QUERY_LP_SETTINGS	The PEP_PPM_QUERY_LP_SETTINGS structure contains a kernel handle to the registry key that contains the power optimization settings that the platform extension plug-in (PEP) has defined for each power scenario.
PEP_PPM_QUERY_PERF_CAPABILITIES	The PEP_PPM_QUERY_PERF_CAPABILITIES structure describes the performance capabilities of the processors in the specified processor performance domain.

TITLE	DESCRIPTION
PEP_PPM_QUERY_PLATFORM_STATE	The PEP_PPM_QUERY_PLATFORM_STATE structure contains information about a platform idle state.
PEP_PPM_QUERY_PLATFORM_STATES	The PEP_PPM_QUERY_PLATFORM_STATES structure specifies the number of platform idle states the hardware platform supports.
PEP_PPM_QUERY_STATE_NAME	The PEP_PPM_QUERY_STATE_NAME structure contains information about a specific coordinated or platform idle state.
PEP_PPM_QUERY_VETO_REASON	The PEP_PPM_QUERY_VETO_REASON structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.
PEP_PPM_QUERY_VETO_REASONS	The PEP_PPM_QUERY_VETO_REASONS structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.
PEP_PPM_RESUME_FROM_SYSTEM_STATE	Used by the PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE notification that notifies the PEP that the system has just resumed from a system power state.
PEP_PPM_TEST_IDLE_STATE	The PEP_PPM_TEST_IDLE_STATE structure contains information about whether the processor can immediately enter a processor idle state.
PEP_PREPARE_DEVICE	The PEP_PREPARE_DEVICE structure identifies a device that must be started up in preparation for its use by the operating system.
PEP_PROCESSOR_FEEDBACK_COUNTER	The PEP_PROCESSOR_FEEDBACK_COUNTER structure describes a feedback counter to the operating system.
PEP_PROCESSOR_IDLE_CONSTRAINTS	The PEP_PROCESSOR_IDLE_CONSTRAINTS structure specifies a set of constraints that the PEP uses to select a processor idle state.
PEP_PROCESSOR_IDLE_DEPENDENCY	The PEP_PROCESSOR_IDLE_DEPENDENCY structure specifies the dependencies of a platform idle state on the specified processor.
PEP_PROCESSOR_IDLE_STATE	The PEP_PROCESSOR_IDLE_STATE structure describes the capabilities of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_UPDATE	The PEP_PROCESSOR_IDLE_STATE_UPDATE structure contains the updated properties of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_V2	The PEP_PROCESSOR_IDLE_STATE_V2 structure describes a processor idle state that the platform extension plug-in (PEP) supports.

TITLE	DESCRIPTION
PEP_PROCESSOR_PARK_PREFERENCE	The PEP_PROCESSOR_PARK_PREFERENCE structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.
PEP_PROCESSOR_PARK_STATE	The PEP_PROCESSOR_PARK_STATE structure describes the parking state for a single processor.
PEP_PROCESSOR_PERF_STATE	Use in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification. This structure describes the properties of a single performance state. .
PEP_QUERY_COMPONENT_PERF_CAPABILITIES	The PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure specifies the number of performance state (P-state) sets that are defined for a component.
PEP_QUERY_COMPONENT_PERF_SET	The PEP_QUERY_COMPONENT_PERF_SET structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_SET_NAME	The PEP_QUERY_COMPONENT_PERF_SET_NAME structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_STATES	The PEP_QUERY_COMPONENT_PERF_STATES structure contains a list of discrete performance state (P-state) values for the specified P-state set.
PEP_QUERY_CURRENT_COMPONENT_PERF_STATE	The PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure contains information about the current P-state in the specified P-state set.
PEP_QUERY_SOC_SUBSYSTEM	The PEP_QUERY_SOC_SUBSYSTEM structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME	The PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM_COUNT	The PEP_QUERY_SOC_SUBSYSTEM_COUNT structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.
PEP_QUERY_SOC_SUBSYSTEM_METADATA	The PEP_QUERY_SOC_SUBSYSTEM_METADATA structure is used with the PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.
PEP_REGISTER_COMPONENT_PERF_STATES	The PEP_REGISTER_COMPONENT_PERF_STATES structure describes the performance states (P-states) of the specified component.

TITLE	DESCRIPTION
PEP_REGISTER_CRASHDUMP_DEVICE	The PEP_REGISTER_CRASHDUMP_DEVICE structure provides a callback routine to turn on a crash-dump device.
PEP_REGISTER_DEBUGGER	The PEP_REGISTER_DEBUGGER structure identifies a registered device that is a core system resource that provides debugger transport.
PEP_REGISTER_DEVICE_V2	The PEP_REGISTER_DEVICE_V2 structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).
PEP_REQUEST_COMPONENT_PERF_STATE	The PEP_REQUEST_COMPONENT_PERF_STATE structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).
PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING	The PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.
PEP_SOC_SUBSYSTEM_METADATA	The PEP_SOC_SUBSYSTEM_METADATA structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).
PEP_SYSTEM_LATENCY	The PEP_SYSTEM_LATENCY structure specifies the new value for the system latency tolerance.
PEP_UNMASKED_INTERRUPT_FLAGS	The PEP_UNMASKED_INTERRUPT_FLAGS union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.
PEP_UNMASKED_INTERRUPT_INFORMATION	The PEP_UNMASKED_INTERRUPT_INFORMATION structure contains information about an interrupt source.
PEP_UNREGISTER_DEVICE	The PEP_UNREGISTER_DEVICE structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).
PEP_WORK	The PEP_WORK structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).
PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE	The PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).
PEP_WORK_ACPI_NOTIFY	The PEP_WORK_ACPI_NOTIFY structure contains the ACPI Notify code for a device that has generated a hardware event.

TITLE	DESCRIPTION
PEP_WORK_ACTIVE_COMPLETE	The PEP_WORK_ACTIVE_COMPLETE structure identifies a component that is now in the active condition.
PEP_WORK_COMPLETE_IDLE_STATE	The PEP_WORK_COMPLETE_IDLE_STATE structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new Fx power state.
PEP_WORK_COMPLETE_PERF_STATE	The PEP_WORK_COMPLETE_PERF_STATE structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.
PEP_WORK_DEVICE_IDLE	The PEP_WORK_DEVICE_IDLE structure indicates whether to ignore the idle time-out for the specified device.
PEP_WORK_DEVICE_POWER	The PEP_WORK_DEVICE_POWER structure describes the new power requirements for the specified device.
PEP_WORK_IDLE_STATE	The PEP_WORK_IDLE_STATE structure contains a request to transition a component to an Fx power state.
PEP_WORK_INFORMATION	The PEP_WORK_INFORMATION structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).
PEP_WORK_POWER_CONTROL	The PEP_WORK_POWER_CONTROL structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.
PO_FX_CORE_DEVICE	The PO_FX_CORE_DEVICE structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.

Enumerations

TITLE	DESCRIPTION
GPIO_PIN_CONFIG_TYPE	The GPIO_PIN_CONFIG_TYPE enumeration describes a connection IO resource.
GPIO_PIN_IORESTRICTION_TYPE	The GPIO_PIN_IORESTRICTION_TYPE enumeration describes the functions that a GPIO pin is limited to performing.
PEP_ACPI_OBJECT_TYPE	The PEP_ACPI_OBJECT_TYPE enumeration indicates the type of ACPI object.
PEP_ACPI_RESOURCE_TYPE	The PEP_ACPI_RESOURCE_TYPE enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.
PEP_DEVICE_ACCEPTANCE_TYPE	The PEP_DEVICE_ACCEPTANCE_TYPE enumeration indicates whether a PEP accepts ownership of a device.

TITLE	DESCRIPTION
PEP_PERF_STATE_TYPE	The PEP_PERF_STATE_TYPE enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.
PEP_PERF_STATE_UNIT	The PEP_PERF_STATE_UNIT enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.
PEP_PROCESSOR_IDLE_CANCEL_CODE	The PEP_PROCESSOR_IDLE_CANCEL_CODE enumeration values indicate reasons why a processor cannot enter an idle state that was previously selected by the platform extension plug-in (PEP).
PEP_PROCESSOR_IDLE_TYPE	The PEP_PROCESSOR_IDLE_TYPE enumeration indicates whether idle constraints apply to just the current processor or to all processors in the hardware platform.
PEP_WORK_TYPE	The PEP_WORK_TYPE enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.

PEP_PROCESSOR_IDLE_TYPE Enumeration

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_TYPE** enumeration indicates whether idle constraints apply to just the current processor or to all processors in the hardware platform.

Syntax

```
typedef enum  {
  PepIdleTypeProcessor,
  PepIdleTypePlatform,
  PepIdleTypeMax
} PEP_PROCESSOR_IDLE_TYPE, *PPEP_PROCESSOR_IDLE_TYPE;
```

Constants

PepIdleTypeProcessor	Apply to current processor.
PepIdleTypePlatform	Apply to all processors.
PepIdleTypeMax	Reserved for use by operating system.

Remarks

The **Type** member of the PEP_PROCESSOR_IDLE_CONSTRAINTS structure is a **PEP_PROCESSOR_IDLE_TYPE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_PROCESSOR_IDLE_CONSTRAINTS

PEP_PROCESSOR_IDLE_CANCEL_CODE Enumeration

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_CANCEL_CODE** enumeration values indicate reasons why a processor cannot enter an idle state that was previously selected by the platform extension plug-in (PEP).

Syntax

```
typedef enum  {
  PepIdleCancelWorkPending,
  PepIdleCancelDependencyCheckFailed,
  PepIdleCancelNoCState,
  PepIdleCancelMax
} PEP_PROCESSOR_IDLE_CANCEL_CODE, *PPEP_PROCESSOR_IDLE_CANCEL_CODE;
```

Constants

PepIdleCancelWorkPending	The processor has pending work that prevents it from entering the selected idle state.
PepIdleCancelDependencyCheckFailed	The processor can enter the selected idle state only after one or more secondary processors have entered their corresponding idle states, but not all of these secondary processors have entered the correct idle states.
PepIdleCancelNoCState	The selected idle state corresponds to a C-state that is not supported. The PEP previously received a PEP_NOTIFY_PPM_CST_STATES notification that supplied a list of the supported C-states for this processor.
PepIdleCancelMax	Reserved for use by the operating system.

Remarks

The **CancelCode** member of the PEP_PPM_IDLE_CANCEL structure contains a **PEP_PROCESSOR_IDLE_CANCEL_CODE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_NOTIFY_PPM_CST_STATES

PEP_PPM_IDLE_CANCEL

GPIO_PIN_CONFIG_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **GPIO_PIN_CONFIG_TYPE** enumeration describes a connection IO resource.

Syntax

```
typedef enum _GPIO_PIN_CONFIG_TYPE {
  PullDefault,
  PullUp,
  PullDown,
  PullNone
} GPIO_PIN_CONFIG_TYPE;
```

Constants

PullDefault	Indicates that no configuration is applied to this pin.
PullUp	Indicates that this pin is configured to use a pull-up resistor.
PullDown	Indicates that this pin is configured to use a pull-down resistor.
PullNone	Indicates that this pin is not configured to use a pull-up or pull-down resistor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

GPIO_PIN_IORESTRICTION_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **GPIO_PIN_IORESTRICTION_TYPE** enumeration describes the functions that a GPIO pin is limited to performing.

Syntax

```
typedef enum _GPIO_PIN_IORESTRICTION_TYPE {
  IoRestrictionNone,
  IoRestrictionInputOnly,
  IoRestrictionOutputOnly,
  IoRestrictionNoneAndPreserve
} GPIO_PIN_IORESTRICTION_TYPE;
```

Constants

IoRestrictionNone	Indicates that the GPIO pin is not restricted to either input or output. When no IO restriction is described, it is assumed to be **IoRestrictionNone**.
IoRestrictionInputOnly	Indicates that the GPIO pin is restricted to input.
IoRestrictionOutputOnly	Indicates that the GPIO pin is restricted to output.
IoRestrictionNoneAndPreserve	Indicates that the GPIO pin is not restricted to either input or output and that the mode should be preserved when the driver is unloaded.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

PEP_ABANDON_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ABANDON_DEVICE** structure identifies a device that has been abandoned and will no longer be used by the operating system.

Syntax

```
typedef struct _PEP_ABANDON_DEVICE {
  PCUNICODE_STRING DeviceId;
  BOOLEAN          DeviceAccepted;
} PEP_ABANDON_DEVICE, *PPEP_ABANDON_DEVICE;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device. The PEP sets this member to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device.

Remarks

This structure is used by the PEP_DPM_ABANDON_DEVICE notification. The **DeviceId** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_ABANDON_DEVICE

UNICODE_STRING

PEP_ACPI_ABANDON_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_ABANDON_DEVICE** structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.

Syntax

```
typedef struct _PEP_ACPI_ABANDON_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  BOOLEAN          DeviceAccepted;
} PEP_ACPI_ABANDON_DEVICE, *PPEP_ACPI_ABANDON_DEVICE;
```

Members

`AcpiDeviceName`

[in] A pointer to an ANSI_STRING structure that contains the fully qualified BIOS name for the device. This name specifies the the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device specified by the **AcpiDeviceName** member. Set to TRUE if the PEP claims ownership, and to FALSE if the PEP does not own the device.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_ABANDON_DEVICE notification. This notification provides an opportunity for the PEP to clean up any remaining device state after the operating system has abandoned the device. The **AcpiDeviceName** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_NOTIFY_ACPI_ABANDON_DEVICE

PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure contains an enumeration of the objects in the namespace of the device.

Syntax

```
typedef struct _PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE {
  PEPHANDLE                       DeviceHandle;
  ULONG                           RequestFlags;
  NTSTATUS                        Status;
  ULONG                           ObjectCount;
  SIZE_T                          ObjectBufferSize;
  PEP_ACPI_OBJECT_NAME_WITH_TYPE Objects[ANYSIZE_ARRAY];
} PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE, *PPEP_ACPI_ENUMERATE_DEVICE_NAMESPACE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_EDN_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of the requested device-namespace enumeration. Set this member to STATUS_SUCCESS if the PEP succeeds in enumerating the objects in the device namespace. Set to STATUS_BUFFER_TOO_SMALL if the input value in the **TotalBufferSize** member indicates that the allocated buffer is not large enough to contain the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure and **Objects** array.

`ObjectCount`

[out] The number of elements in the **Objects** array.

`ObjectBufferSize`

`Objects`

[out] The first element in an array of PEP_ACPI_OBJECT_NAME_WITH_TYPE structures that describe the objects in the ACPI namespace of the device. If this array contains more than one element, the additional array elements follow the end of the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification. The

RequestFlags and **TotalBufferSize** members contain input values supplied by PoFx when the notification is sent. The PEP may overwrite the **TotalBufferSize** input value with an output value if the input value is less than the required output buffer size. The **Status**, **ObjectCount**, and **Objects** members contain output values that the PEP writes to the structure.

If N is the number of objects listed in the ACPI namespace under this device, the PEP should verify that the storage allocated by PoFx for the **Objects** array is large enough to contain N array elements. If N > 1, the additional N−1 array elements follow the end of the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure. The buffer size required to contain the structure and additional array elements is calculated as follows:

sizeof PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE sizeof PEP_ACPI_OBJECT_NAME_WITH_TYPE If the input value specified in the **TotalBufferSize** member is less than the required size, the PEP overwrites the **TotalBufferSize** input value with the required size, and sets the **Status** member to STATUS_BUFFER_TOO_SMALL. In response, PoFx will allocate a buffer of the required size and send a second PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification to the PEP for this device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME_WITH_TYPE

PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_EVALUATE_CONTROL_METHOD structure

2/5/2019 • 3 minutes to read • Edit Online

The **PEP_ACPI_EVALUATE_CONTROL_METHOD** structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.

Syntax

```
typedef struct _PEP_ACPI_EVALUATE_CONTROL_METHOD {
  PEPHANDLE              DeviceHandle;
  ULONG                  RequestFlags;
  union {
    ULONG       MethodName;
    ANSI_STRING MethodNameString;
  };
  NTSTATUS               MethodStatus;
  PVOID                  CompletionContext;
  ULONG                  InputArgumentCount;
  SIZE_T                 InputArgumentSize;
  PACPI_METHOD_ARGUMENT InputArguments;
  ULONG                  OutputArgumentCount;
  SIZE_T                 OutputArgumentSize;
  PACPI_METHOD_ARGUMENT OutputArguments;
} PEP_ACPI_EVALUATE_CONTROL_METHOD, *PPEP_ACPI_EVALUATE_CONTROL_METHOD;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of flags. This member contains one of the following values.

FLAG NAME	VALUE	MEANING
PEP_ACPI_ECM_FLAG_NONE	0x0	Not a valid control method name.
PEP_ACPI_ECM_FLAG_RELATIVE_NAME	0x1	The **MethodNameUlong** member contains a path-relative control method name.
PEP_ACPI_ECM_FLAG_FULLY_QUALIFIED_NAME	0x2	The **MethodNameString** member contains a fully qualified control method name.

`MethodName`

`MethodNameString`

`MethodStatus`

[out] An NTSTATUS value that indicates the status of the evaluation of the ACPI control method. Set this member

to STATUS_SUCCESS to indicate that the platform extension plug-in (PEP) successfully completed its evaluation of the method. Set to STATUS_NOT_SUPPORTED to indicate that the PEP does not recognize the control method. Set to STATUS_BUFFER_TOO_SMALL to indicate that the output buffer size specified by the **InputArgumentSize** member is not large enough to contain the method results.

If the PEP is to evaluate the method asychronously, set this member to STATUS_PENDING and return from the AcceptAcpiNotification callback. Later, when the evaluation is completed, the PEP calls the CompleteWork routine to notify the Windows power management framework (PoFx) that the evaluation of the control method is complete.

`CompletionContext`

[in] A pointer to a completion context value. The PEP uses this member only if the control method is evaluated asynchronously. In this case, the PEP supplies this completion context pointer in the call to the CompleteWork routine that notifies the Windows power management framework (PoFx) that the evaluation of the control method is complete. Included in the input parameters to this call is a pointer to a PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure to which the PEP has written the completion context pointer. The context is opaque to the PEP and contains data used internally by PoFx.

`InputArgumentCount`

[in] The number of input arguments in the input buffer pointed to by the **InputArguments** member. Each argument starts with an ACPI_METHOD_ARGUMENT structure that specifies the size of the argument. Currently, the maximum **InputArgumentCount** value is one.

`InputArgumentSize`

[in] The size, in bytes, of the input buffer pointed to by the **InputArguments** member.

`InputArguments`

[in] A pointer to an input buffer that contains an ACPI_METHOD_ARGUMENT structure that specifies the input argument for the specified ACPI control method.

`OutputArgumentCount`

[in] The number of output arguments in the output buffer pointed to by the **OutputArguments** member. Each argument starts with an ACPI_METHOD_ARGUMENT structure that specifies the size of the argument. Currently, the maximum allowed **OutputArgumentCount** value is one.

`OutputArgumentSize`

[in, out] On input, the size, in bytes, of the output buffer pointed to by the **OutputArguments** member. If this buffer is not large enough to contain the result of evaluating the method, the PEP overwrites the **OutputArgumentSize** input value with the required size, and sets the **MethodStatus** member to STATUS_BUFFER_TOO_SMALL.

`OutputArguments`

[in] A pointer to an output buffer to which the PEP writes an ACPI_METHOD_ARGUMENT structure that contains the result of evaluating the specified ACPI control method.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification. The **MethodStatus** member contains an output value that the PEP writes to the structure in response to this notification. The **OutputArgumentSize** member contains an input value supplied by PoFx when the notification is sent. The PEP may overwrite this input value with an output value if the input value is less than the required output buffer size. All other members of this structure contain input values that are supplied by PoFx when the notification

is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

ANSI_STRING

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_EXTENDED_ADDRESS structure

1/30/2019 • 5 minutes to read • Edit Online

The **PEP_ACPI_EXTENDED_ADDRESS** structure is used to report resource usage in the address space such as memory and IO.

Syntax

```
typedef struct _PEP_ACPI_EXTENDED_ADDRESS {
  PEP_ACPI_RESOURCE_TYPE  Type;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  UCHAR                   ResourceFlags;
  UCHAR                   GeneralFlags;
  UCHAR                   TypeSpecificFlags;
  UCHAR                   RevisionId;
  UCHAR                   Reserved;
  ULONGLONG               Granularity;
  ULONGLONG               MinimumAddress;
  ULONGLONG               MaximumAddress;
  ULONGLONG               TranslationAddress;
  ULONGLONG               AddressLength;
  ULONGLONG               TypeAttribute;
  PUNICODE_STRING         DescriptorName;
} PEP_ACPI_EXTENDED_ADDRESS, *PPEP_ACPI_EXTENDED_ADDRESS;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure describing this resource.

`ResourceFlags`

Indicates the type of resource this structure describes.

VALUE	MEANING
0	Indicates that this resource is a memory range.
1	Indicates that this resource is an IO range.
2	Indicates that this resource is a bus number range.
3-191	These values are reserved for future use.

192 - 255	These values are reserved for use by the hardware vendor.

`GeneralFlags`

A value containing the bit flags that are common to all resource types.

BIT(S)	MEANING
0 - Consumer/ Producer flag	When set, this indicates that the device consumes this resource.
1 - Decode type	When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only. When not set, indicates that this bridge positively decodes this address.
2 - Minimum address fixed	When set, indicates that the minimum address is fixed.
3 - Max address fixed	When set, indicates that the maximum address is fixed.
4 to 7 - Reserved	These bits are reserved and must be set to zero.

`TypeSpecificFlags`

The value of this member is dependent on the value in **ResourceFlags** member. The flags for each resource type are described in the tables below.

Memory Resource (**ResourceFlags** = **0**)

BIT(S)	MEANING
0 - Write status	When set, indicates that this memory range is available for reading and writing. Otherwise, this indicates that this memory range is read-only.
1 to 2 - Memory attributes (MEM)	**0** - Indicates the memory is non-cacheable. **1** - Indicates the memory is cacheable. **2** - Indicates the memory is cacheable and supports write combining. **3** - The memory is cacheable and prefetchable.
3 to 4 - Memory attributes (MTP)	These bits are only defined if this memory resource describes system RAM. **0** - Address range memory: This range is available RAM usable by the operating system. **1** - Address range reserved: This range of addresses is in use or reserved by the system and is not to be included in the allocatable memory pool of the operating system's memory manager. **2** - Address range ACPI: ACPI Reclaim Memory. This range is available RAM usable by the OS after it reads the ACPI tables. **3** - Address Range NVS: ACPI NVS Memory. This range of addresses is in use or reserved by the system and must not be used by the operating system. This range is required to be saved and restored across an NVS sleep.
5 - Memory to IO translation	**0** - Type-static: This resource is memory on the primary and secondary sides of the bridge. **1** - Type-translation: This resource is memory on the secondary side of the bridge and IO on the primary side of the bridge.
6 to 7 - Reserved	These bits are reserved and must be set to zero.

IO Resource (**ResourceFlags** = **1**)

BIT(S)	MEANING

0 to 1 - Range	**0** - Reserved. **1** - Non-ISA ranges only. This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the non-ISA IO addresses that fall within the specified window. The non-ISA IO ranges are: n100-n3FF, n500-n7FF, n900-nBFF, nD00-nFFF. This bit can only be set for bridges entirely configured through the ACPI namespace. **2** - ISA ranges only. This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the ISA IO addresses that fall within the specified window. The ISA IO ranges are: n000-n0FF, n400-n4FF, n800-n8FF, nC00-nCFF. This bit can only be set for bridges entirely configured through the ACPI namespace. **3** - The memory window covers the entire range
2 to 3 - Reserved	These bits are reserved and must be set to zero.
4 - IO to memory translation	**0** - Type-static: This resource is IO on the primary and secondary sides of the bridge. **1** - Type-translation: This resource is IO on the secondary side of the bridge and memory on the primary side of the bridge.
5 - Sparse translation	This bit is only meaningful if bit 4 (IO to memory translation) is set. **0** - Dense translation: The primary-side memory address of any specific IO port within the secondary-side range can be found using the following function. *address = port + TranslationAddress* **1** - Sparse translation: The primary-side memory address of any specific IO port within the secondary-side range can be found using the following function. *address = (((port & 0xFFFc) << 10) || (port & 0xFFF)) + TranslationAddress* In the address used to access the IO port, bits 2 to 11 must be identical to bits 12 to 21, this gives four bytes of IO ports on each 4 KB page.
6 to 7 - Reserved	These bits are reserved and must be set to zero.

Bus Number Range Resource (**ResourceFlags** = **2**)

BIT(S)	MEANING

0 to 7 - Reserved	These bits are reserved and must be set to zero.

`RevisionId`

Indicates the revision of the extended address space descriptor detailed by this structure. For ACPI 3.0, this value is 1.

`Reserved`

`Granularity`

A bit mask indicating which bits have been decoded.

`MinimumAddress`

The minimum starting address. For bridges that translate addresses, this is the address space on the secondary side of the bridge.

`MaximumAddress`

The maximum starting address. For bridges that translate addresses, this is the address space on the secondary side of the bridge.

`TranslationAddress`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`AddressLength`

The address length.

`TypeAttribute`

Indicates attributes that are specific to the resource type that is specified in the **ResourceFlags** member. If **ResourceFlags** is zero, this value is zero, otherwise the meaning of the value can be found in the *UEFI Specification* in the section titled **GetMemoryMap()**.

`DescriptorName`

The name of this resource descriptor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

PEP_ACPI_GPIO_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_GPIO_RESOURCE** structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.

Syntax

```
typedef struct _PEP_ACPI_GPIO_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE      Type;
  PEP_ACPI_RESOURCE_FLAGS     Flags;
  KINTERRUPT_MODE             InterruptType;
  KINTERRUPT_POLARITY         InterruptPolarity;
  GPIO_PIN_CONFIG_TYPE        PinConfig;
  GPIO_PIN_IORESTRICTION_TYPE IoRestrictionType;
  USHORT                      DriveStrength;
  USHORT                      DebounceTimeout;
  PUSHORT                     PinTable;
  USHORT                      PinCount;
  UCHAR                       ResourceSourceIndex;
  PUNICODE_STRING             ResourceSourceName;
  PUCHAR                      VendorData;
  USHORT                      VendorDataLength;
} PEP_ACPI_GPIO_RESOURCE, *PPEP_ACPI_GPIO_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value that identifies the resource type for this ACPI resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptPolarity`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`IoRestrictionType`

A GPIO_PIN_IORESTRICTION_TYPE enumeration value that identifies the type of IO that the pin supports.

`DriveStrength`

Specifies the output drive capability of the pin, in hundredths of milliamperes.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`PinTable`

A list of pin numbers on the resource that are described by this descriptor.

`PinCount`

The number of pins in **PinTable**.

`ResourceSourceIndex`

This member is always zero.

`ResourceSourceName`

This member is always set to "ResourceConsumer."

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the **VendorData** buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

GPIO_PIN_IORESTRICTION_TYPE

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_INTERRUPT_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INTERRUPT_RESOURCE** structure describes an ACPI interrupt resource.

Syntax

```
typedef struct _PEP_ACPI_INTERRUPT_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE  Type;
  KINTERRUPT_MODE         InterruptType;
  KINTERRUPT_POLARITY     InterruptPolarity;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  UCHAR                   Count;
  PULONG                  Pins;
} PEP_ACPI_INTERRUPT_RESOURCE, *PPEP_ACPI_INTERRUPT_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptPolarity`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`Count`

The count of items in the **Pins** array.

`Pins`

A list of pin numbers on the resource that are described by this descriptor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_IO_MEMORY_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_IO_MEMORY_RESOURCE** structure describes an ACPI IO port descriptor resource.

Syntax

```
typedef struct _PEP_ACPI_IO_MEMORY_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE Type;
  UCHAR                  Information;
  PHYSICAL_ADDRESS       MinimumAddress;
  PHYSICAL_ADDRESS       MaximumAddress;
  ULONG                  Alignment;
  ULONG                  Length;
} PEP_ACPI_IO_MEMORY_RESOURCE, *PPEP_ACPI_IO_MEMORY_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value that identifies the resource type for this ACPI resource.

`Information`

If bit 0 is a 1, this indicates that the logical device decodes 16-bit addresses. If bit 0 is 0, this indicates that the logical device only decodes the first 10 bits of the address.

Bits 1 to 7 of this member are reserved and must be set to zero.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the IO address assigned.

`Length`

Specifies the number of bytes in the IO range.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_OBJECT_NAME union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_NAME** union contains the four-character name of an ACPI object.

Syntax

```
typedef union _PEP_ACPI_OBJECT_NAME {
  UCHAR Name[4];
  ULONG NameAsUlong;
} PEP_ACPI_OBJECT_NAME, *PPEP_ACPI_OBJECT_NAME;
```

Members

`Name`

`NameAsUlong`

Remarks

The *Name* member of the PEP_ACPI_OBJECT_NAME_WITH_TYPE structure is a **PEP_ACPI_OBJECT_NAME** union. Also, the *Name* member of the PEP_ACPI_QUERY_OBJECT_INFORMATION structure is a **PEP_ACPI_OBJECT_NAME** union.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME_WITH_TYPE

PEP_ACPI_QUERY_OBJECT_INFORMATION

PEP_ACPI_OBJECT_NAME_WITH_TYPE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_NAME_WITH_TYPE** structure that specifies both the path-relative name of an ACPI object and the type of this object.

Syntax

```
typedef struct _PEP_ACPI_OBJECT_NAME_WITH_TYPE {
  PEP_ACPI_OBJECT_NAME Name;
  PEP_ACPI_OBJECT_TYPE Type;
} PEP_ACPI_OBJECT_NAME_WITH_TYPE, *PPEP_ACPI_OBJECT_NAME_WITH_TYPE;
```

Members

`Name`

A PEP_ACPI_OBJECT_NAME structure that contains the four-character name of the ACPI object.

`Type`

A PEP_ACPI_OBJECT_TYPE enumeration value that indicates the type of ACPI object. Currently, the only supported object type is a control method.

Remarks

The **Objects** member of the PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure is the first element in an array of **PEP_ACPI_OBJECT_NAME_WITH_TYPE** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE

PEP_ACPI_OBJECT_NAME

PEP_ACPI_OBJECT_TYPE

PEP_ACPI_OBJECT_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_TYPE** enumeration indicates the type of ACPI object.

Syntax

```
typedef enum _PEP_ACPI_OBJECT_TYPE {
  PepAcpiObjectTypeMethod,
  PepAcpiObjectTypeDevice,
  PepAcpiObjectTypeMaximum
} PEP_ACPI_OBJECT_TYPE, *PPEP_ACPI_OBJECT_TYPE;
```

Constants

PepAcpiObjectTypeMethod	The object is an ACPI control method.
PepAcpiObjectTypeDevice	
PepAcpiObjectTypeMaximum	Reserved for use by the operating system.

Remarks

The **Type** member of the PEP_ACPI_QUERY_OBJECT_INFORMATION structure is an **PEP_ACPI_OBJECT_TYPE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_QUERY_OBJECT_INFORMATION

PEP_ACPI_PREPARE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_PREPARE_DEVICE** structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.

Syntax

```
typedef struct _PEP_ACPI_PREPARE_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  ULONG            InputFlags;
  BOOLEAN          DeviceAccepted;
  ULONG            OutputFlags;
} PEP_ACPI_PREPARE_DEVICE, *PPEP_ACPI_PREPARE_DEVICE;
```

Members

`AcpiDeviceName`

[in] A pointer to an ANSI_STRING structure that contains the fully qualified BIOS name for the device. This name specifies the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_PREPARE_DEVICE_INPUT_FLAG_NONE (0x0).

`DeviceAccepted`

[out] Whether the PEP is prepared to provide ACPI services for the device. Set this member to TRUE if the PEP is prepared to be the sole provider of ACPI services for this device, and to FALSE if it is not.

`OutputFlags`

[out] A set of output flags. No flags are currently defined for this member. Set this member to PEP_ACPI_PREPARE_DEVICE_OUTPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_PREPARE_DEVICE notification. The **AcpiDeviceName** and **InputFlags** members of the structure contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **DeviceAccepted** and **OutputFlags** members contain output values that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_NOTIFY_ACPI_PREPARE_DEVICE

PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES** structure contains a list of raw resources that are needed to control power to the device.

Syntax

```
typedef struct _PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES {
  PEPHANDLE            DeviceHandle;
  ULONG                RequestFlags;
  NTSTATUS             Status;
  SIZE_T               BiosResourcesSize;
  ACPI_METHOD_ARGUMENT BiosResources[ANYSIZE_ARRAY];
} PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES, *PPEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_QDCR_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of this query. Set this member to STATUS_SUCCESS if the PEP succeeded in providing the requested list of resources. Set to STATUS_BUFFER_TOO_SMALL to indicate that the output buffer is not large enough to contain the resource list.

`BiosResourcesSize`

[in, out] On input, the size, in bytes, of the buffer allocated for the **BiosResources** member. This buffer size includes any associated data that might follow this member.

If this buffer is not large enough to contain the **ACPI_METHOD_ARGUMENT** structure and its associated data, the PEP overwrites the input value of **BiosResourcesSize** with the required size and sets the **Status** member to STATUS_BUFFER_TOO_SMALL.

`BiosResources`

[in] An output buffer to which the PEP writes an ACPI_METHOD_ARGUMENT structure that contains the requested resource list. The **Data** member of this structure is the first member in an array that might extend past the end of the structure. The buffer size specified by **BiosResourcesSize** includes both the **ACPI_METHOD_ARGUMENT** structure and any **Data** array elements that follow this structure. For more information, see TBD.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES notification. The **RequestFlags**, **BiosResourcesSize**, and **BiosResources** members of the structure contain input values supplied by the Windows power management framework (PoFx) when this notification is sent. If the **BiosResourcesSize** input value is too small, the PEP should overwrite this value. The **Status** member contains an output value that the PEP writes to this structure in response to the notification.

If the input value specified in the **BiosResourcesSize** member is less than the required size, the PEP overwrites the **BiosResourcesSize** input value with the required size, and sets the **Status** member to STATUS_BUFFER_TOO_SMALL. In response, PoFx will try to allocate a buffer of the required size and send a second **PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES** notification to the PEP for this device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_QUERY_OBJECT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_QUERY_OBJECT_INFORMATION** structure contains information about an ACPI object.

Syntax

```
typedef struct _PEP_ACPI_QUERY_OBJECT_INFORMATION {
  PEPHANDLE            DeviceHandle;
  PEP_ACPI_OBJECT_NAME Name;
  PEP_ACPI_OBJECT_TYPE Type;
  ULONG                ObjectFlags;
  union {
    struct {
      ULONG InputArgumentCount;
      ULONG OutputArgumentCount;
    } MethodObject;
  } DUMMYUNIONNAME;
} PEP_ACPI_QUERY_OBJECT_INFORMATION, *PPEP_ACPI_QUERY_OBJECT_INFORMATION;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`Name`

[in] A PEP_ACPI_OBJECT_NAME union that specifies the path-relative, four-character name of the object.

`Type`

[in] A PEP_ACPI_OBJECT_TYPE enumeration value that specifies the object type. Currently, a control method is the only object type that is defined for this member (**Type** = **PepAcpiObjectTypeMethod**).

`ObjectFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_OBJECT_FLAG_NONE (0x0).

`DUMMYUNIONNAME`

The query result. If the specified object is a control method, the platform extension plug-in (PEP) writes the query result to the **MethodObject** member of this union.

`DUMMYUNIONNAME.MethodObject`

`DUMMYUNIONNAME.MethodObject.InputArgumentCount`

The number of input arguments expected by the control method.

`DUMMYUNIONNAME.MethodObject.OutputArgumentCount`

The number of output arguments produced by the control method.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION notification. The **Name**, **Type**, and **Flags** members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent. The **MethodObject** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME

PEP_ACPI_OBJECT_TYPE

PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_REGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_REGISTER_DEVICE** structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.

Syntax

```
typedef struct _PEP_ACPI_REGISTER_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  ULONG            InputFlags;
  POHANDLE         KernelHandle;
  PEPHANDLE        DeviceHandle;
  ULONG            OutputFlags;
} PEP_ACPI_REGISTER_DEVICE, *PPEP_ACPI_REGISTER_DEVICE;
```

Members

`AcpiDeviceName`

[in] An ANSI_STRING structure that contains the fully qualified BIOS name for the device. The same name was previously supplied as an input value in the PEP_ACPI_PREPARE_DEVICE notification for the device. This name specifies the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_REGISTER_DEVICE_INPUT_FLAG_NONE (0x0).

`KernelHandle`

[in] A POHANDLE value that represents the registration of the device with the Windows power management framework (PoFx). The PEP can use this handle to identify the device in subsequent communications with PoFx.

`DeviceHandle`

[out] A PEPHANDLE value that identifies the registration of this device for ACPI services. This handle is created by the PEP. In subsequent ACPI service notifications, PoFx will use this handle to identify the device.

`OutputFlags`

[out] A set of output flags. No flags are currently defined for this member. Set this member to PEP_ACPI_REGISTER_DEVICE_OUTPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification. The **AcpiDeviceName**, **InputFlags**, and **KernelHandle** members of the structure contain input values that are supplied by PoFx when this notification is sent to the PEP. The **DeviceHandle** and **OutputFlags** members contains output values that the PEP writes to this structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_ACPI_PREPARE_DEVICE

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES** structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.

Syntax

```
typedef struct _PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES {
  NTSTATUS           TranslationStatus;
  PPEP_ACPI_RESOURCE InputBuffer;
  SIZE_T             InputBufferSize;
  PVOID              OutputBuffer;
  SIZE_T             OutputBufferSize;
  ULONG              Flags;
} PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES, *PPEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES;
```

Members

`TranslationStatus`

The result code from the resource translation call.

`InputBuffer`

A pointer to a PEP_ACPI_RESOURCE containing the input buffer.

`InputBufferSize`

The size of the input buffer.

`OutputBuffer`

A pointer to the output buffer containing the translated structure.

`OutputBufferSize`

The size of the output buffer.

`Flags`

The value contained in the PEP_ACPI_RESOURCE_FLAGS structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP initialization functions

PEP_ACPI_RESOURCE

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE** structure contains hardware details for a specific ACPI resource.

Syntax

```
typedef union _PEP_ACPI_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE      Type;
  PEP_ACPI_IO_MEMORY_RESOURCE IoMemory;
  PEP_ACPI_INTERRUPT_RESOURCE Interrupt;
  PEP_ACPI_GPIO_RESOURCE      Gpio;
  PEP_ACPI_SPB_I2C_RESOURCE   SpbI2c;
  PEP_ACPI_SPB_SPI_RESOURCE   SpbSpi;
  PEP_ACPI_SPB_UART_RESOURCE  SpbUart;
  PEP_ACPI_EXTENDED_ADDRESS   ExtendedAddress;
} PEP_ACPI_RESOURCE, *PPEP_ACPI_RESOURCE;
```

Members

`Type`

The PEP_ACPI_RESOURCE_TYPE enumeration value that is applicable to this resource.

`IoMemory`

If **Type** is **PepAcpiMemory** or **PepAcpiIoPort**, this union contains a PEP_ACPI_IO_MEMORY_RESOURCE structure describing an ACPI IO or memory resource.

`Interrupt`

If **Type** is **PepAcpiInterrupt**, this union contains a PEP_ACPI_INTERRUPT_RESOURCE structure describing an ACPI interrupt resource.

`Gpio`

If **Type** is **PepAcpiGpioIo** or **PepAcpiGpioInt**, this union contains a PEP_ACPI_GPIO_RESOURCE structure describing an ACPI GPIO resource.

`SpbI2c`

If **Type** is **PepAcpiSpbI2c**, this union contains a PEP_ACPI_SPB_I2C_RESOURCE structure describing an ACPI I2C serial bus resource.

`SpbSpi`

If **Type** is **PepAcpiSpbSpi**, this union contains a PEP_ACPI_SPB_SPI_RESOURCE structure describing an ACPI SPI serial bus resource.

`SpbUart`

If **Type** is **PepAcpiSpbUart**, this union contains a PEP_ACPI_SPB_UART_RESOURCE structure describing an ACPI SPB serial bus resource.

`ExtendedAddress`

If **Type** is **PepAcpiExtendedMemory** or **PepAcpiExtendedIo**, this union contains a PEP_ACPI_EXTENDED_ADDRESS structure describing an ACPI extended memory or extended IO resource.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INTERRUPT_RESOURCE

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_SPB_I2C_RESOURCE

PEP_ACPI_SPB_SPI_RESOURCE

PEP_ACPI_SPB_UART_RESOURCE

PEP_ACPI_RESOURCE_FLAGS union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE_FLAGS** structure contains flags describing an ACPI resource.

Syntax

```
typedef union _PEP_ACPI_RESOURCE_FLAGS {
  ULONG  AsULong;
  struct {
    ULONG Shared : 1;
    ULONG Wake : 1;
    ULONG ResourceUsage : 1;
    ULONG SlaveMode : 1;
    ULONG AddressingMode : 1;
    ULONG SharedMode : 1;
    ULONG Reserved : 26;
  } DUMMYSTRUCTNAME;
} PEP_ACPI_RESOURCE_FLAGS, *PPEP_ACPI_RESOURCE_FLAGS;
```

Members

`AsULong`

The consolidated values of the flags in **DUMMYSTRUCTNAME**.

`DUMMYSTRUCTNAME`

A structure containing ACPI resource flags.

`DUMMYSTRUCTNAME.Shared`

When set, indicates that this is a shared device.

`DUMMYSTRUCTNAME.Wake`

When set, indicates that this device can be woken from a low-power state.

`DUMMYSTRUCTNAME.ResourceUsage`

When set, indicates that this device is in use.

`DUMMYSTRUCTNAME.SlaveMode`

When set, indicates that this device is in slave mode.

`DUMMYSTRUCTNAME.AddressingMode`

When set, indicates that this device is in addressing mode.

`DUMMYSTRUCTNAME.SharedMode`

When set, indicates that this device is in shared mode.

`DUMMYSTRUCTNAME.Reserved`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

PEP_ACPI_RESOURCE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE_TYPE** enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.

Syntax

```
typedef enum _PEP_ACPI_RESOURCE_TYPE {
  PepAcpiMemory,
  PepAcpiIoPort,
  PepAcpiInterrupt,
  PepAcpiGpioIo,
  PepAcpiGpioInt,
  PepAcpiSpbI2c,
  PepAcpiSpbSpi,
  PepAcpiSpbUart,
  PepAcpiExtendedMemory,
  PepAcpiExtendedIo
} PEP_ACPI_RESOURCE_TYPE;
```

Constants

PepAcpiMemory	Indicates that the resource is an ACPI memory resource.
PepAcpiIoPort	Indicates that the resource is an ACPI IO port resource.
PepAcpiInterrupt	Indicates that the resource is an ACPI interrupt resource.
PepAcpiGpioIo	Indicates that the resource is an ACPI GPIO resource.
PepAcpiGpioInt	Indicates that the resource is an ACPI GPIO interrupt resource.
PepAcpiSpbI2c	Indicates that the resource is an ACPI I2C serial bus resource.
PepAcpiSpbSpi	Indicates that the resource is an ACPI SPI serial bus resource.
PepAcpiSpbUart	Indicates that the resource is an ACPI UART serial bus resource.
PepAcpiExtendedMemory	Indicates that the resource is an ACPI extended memory resource.
PepAcpiExtendedIo	Indicates that the resource is an ACPI extended IO resource.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES

PEP_ACPI_RESOURCE

PEP_ACPI_SPB_I2C_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_I2C_RESOURCE** structure describes an ACPI I2C serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_I2C_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 ConnectionSpeed;
  USHORT                SlaveAddress;
} PEP_ACPI_SPB_I2C_RESOURCE, *PPEP_ACPI_SPB_I2C_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`SlaveAddress`

The I2C bus address for this connection.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_RESOURCE

PEP_ACPI_SPB_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_RESOURCE** structure describes an ACPI serial bus connection resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE  Type;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  USHORT                  TypeSpecificFlags;
  UCHAR                   ResourceSourceIndex;
  PUNICODE_STRING         ResourceSourceName;
  PCHAR                   VendorData;
  USHORT                  VendorDataLength;
} PEP_ACPI_SPB_RESOURCE, *PPEP_ACPI_SPB_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`TypeSpecificFlags`

Specifies the bit flags that are common to all serial bus connection types.

BIT(S)	MEANING
0 - Slave mode	**0x0** - Indicates that communication over this connection is initiated by the controller. **0x1** - Indicates that communication over this connection is initiated by the device.
1 - Consumer/producer mode	**0x0** - Indicates that this device produces and consumes this resource. **0x1** - Indicates that this device consumes this resource.

2 to 7 - Reserved	These bits are reserved and must be set to zero.

`ResourceSourceIndex`

`ResourceSourceName`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by **VendorData**.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

PEP_ACPI_SPB_SPI_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_SPI_RESOURCE** structure describes an ACPI SPI serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_SPI_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 ConnectionSpeed;
  UCHAR                 DataBitLength;
  UCHAR                 Phase;
  UCHAR                 Polarity;
  USHORT                DeviceSelection;
} PEP_ACPI_SPB_SPI_RESOURCE, *PPEP_ACPI_SPB_SPI_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`DataBitLength`

The size, in bits, of the smallest unit of transfer.

`Phase`

The phase of the clock pulse on which to capture data.

`Polarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`DeviceSelection`

The device selection value. This value is specific to the device and may refer to a chip-select line, GPIO line, or other line selection mechanism.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_UART_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_UART_RESOURCE** structure describes an ACPI UART serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_UART_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 BaudRate;
  USHORT                RxBufferSize;
  USHORT                TxBufferSize;
  UCHAR                 Parity;
  UCHAR                 LinesInUse;
} PEP_ACPI_SPB_UART_RESOURCE, *PPEP_ACPI_SPB_UART_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`BaudRate`

The baud rate of the connection.

`RxBufferSize`

The maximum receive buffer size, in bytes, that is supported by this connection.

`TxBufferSize`

The maximum transmit buffer size, in bytes, that is supported by this connection.

`Parity`

Indicates the parity of the connection.

VALUE	MEANING
0x0 0	None
0x0 1	Even
0x0 2	Odd

0x03	Mark
0x04	Space

`LinesInUse`

Flag indicating the serial lines that are enabled. A value of 1 in the bit positions indicates that the line is enabled.

BIT	MEANING
0	This bit is reserved and must be set to zero.
1	This bit is reserved and must be set to zero.
2	Data Carrier Detect (DTD)
3	Ring Indicator (RI)
4	Data Set Ready (DSR)
5	Data Terminal Ready (DTR)
6	Clear to Send (CTS)
7	Request to Send (RTS)

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_RESOURCE

PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES** structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.

Syntax

```
typedef struct _PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES {
  PEPHANDLE         DeviceHandle;
  ULONG             RequestFlags;
  NTSTATUS          Status;
  SIZE_T            TranslatedResourcesSize;
  PCM_RESOURCE_LIST TranslatedResources;
} PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES, *PPEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_TDCR_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of the resource translation. The PEP sets this member to STATUS_SUCCESS to indicate that the PEP successfully received the translated resources. Otherwise, the PEP sets this member to an appropriate error status code.

`TranslatedResourcesSize`

`TranslatedResources`

[in] A CM_RESOURCE_LIST structure that serves as the header for the resource list. The remainder of the resource list immediately follows this header.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification to provide the PEP with a list of translated power control resources. The **RequestFlags**, **ResourceBufferSize**, and **TranslatedResources** members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent. The **Status** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

CM_RESOURCE_LIST

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES

PEP_ACPI_UNREGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_UNREGISTER_DEVICE** structure contains information about a device that has been unregistered from ACPI services.

Syntax

```
typedef struct _PEP_ACPI_UNREGISTER_DEVICE {
  PEPHANDLE DeviceHandle;
  ULONG     InputFlags;
} PEP_ACPI_UNREGISTER_DEVICE, *PPEP_ACPI_UNREGISTER_DEVICE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_UNREGISTER_DEVICE_INPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_UNREGISTER_DEVICE notification. The **DeviceHandle** and **InputFlags** members contain input values that are supplied by the Windows power management framework when this notification is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_NOTIFY_ACPI_UNREGISTER_DEVICE

PEP_COMPONENT_ACTIVE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_ACTIVE** structure identifies a component that is making a transition between the idle condition and the active condition.

Syntax

```
typedef struct _PEP_COMPONENT_ACTIVE {
  PEPHANDLE             DeviceHandle;
  ULONG                 Component;
  BOOLEAN               Active;
  PPEP_WORK_INFORMATION WorkInformation;
  BOOLEAN               NeedWork;
} PEP_COMPONENT_ACTIVE, *PPEP_COMPONENT_ACTIVE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Active`

[in] Whether the component is making a transition to the active condition. If TRUE, the component is making a transition from the idle condition to the active condition. If FALSE, the component is making a transition from the active condition to the idle condition.

`WorkInformation`

[out] A pointer to a PEP_WORK_INFORMATION structure that describes the work that the PEP requests in response to this notification. If **NeedWork** is TRUE, **WorkInformation** must point to a valid **PEP_WORK_INFORMATION** structure. If **NeedWork** is FALSE, **WorkInformation** must be NULL.

`NeedWork`

[out] Whether the PEP has a work request to submit in response to this notification. Set to TRUE if the PEP has work to request, or to FALSE if the PEP has no work to request.

Remarks

This structure is used by the PEP_DPM_COMPONENT_ACTIVE notification. The first three members of the structure contain input values that are supplied by the Windows power management framework (PoFx). The last two members contain output values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_COMPONENT_ACTIVE

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_COMPONENT_PERF_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_INFO** structure describes the performance states (P-states) of a component.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_INFO {
  ULONG                  SetCount;
  PEP_COMPONENT_PERF_SET PerfStateSets[ANYSIZE_ARRAY];
} PEP_COMPONENT_PERF_INFO, *PPEP_COMPONENT_PERF_INFO;
```

Members

`SetCount`

The number of elements in the **PerfStateSets** array.

`PerfStateSets`

The first element in an array of PEP_COMPONENT_PERF_SET structures. Each array element describes a P-state of the component. If the array contains more than one element, the additional elements immediately follow the end of the **PEP_COMPONENT_PERF_INFO** structure.

Remarks

This structure is used in conjunction with the PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification. The **PerfStateSets** member of the PEP_REGISTER_COMPONENT_PERF_STATES structure is a **PEP_COMPONENT_PERF_INFO** structure.

The **PEP_COMPONENT_PERF_INFO** structure contains input values that are supplied by the Windows power management framework (PoFx). The platform extension plug-in (PEP) must not write to this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_COMPONENT_PERF_SET

PEP_DPM_REGISTER_COMPONENT_PERF_STATES

PEP_REGISTER_COMPONENT_PERF_STATES

PEP_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_SET** structure describes the performance states (P-states) in a P-state set.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_SET {
  UNICODE_STRING      Name;
  ULONGLONG           Flags;
  PEP_PERF_STATE_UNIT Unit;
  PEP_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG           Count;
      PPEP_PERF_STATE States;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PEP_COMPONENT_PERF_SET, *PPEP_COMPONENT_PERF_SET;
```

Members

`Name`

An optional string that describes the device property controlled by this P-state set. For example, this string might be "Clock frequency" or "Memory bandwidth". If no such string is available for this P-state, the **Name** member is set to NULL. Otherwise, this member contains a pointer to a UNICODE_STRING structure that contains the string.

`Flags`

Set to zero. No flag bits are currently defined for this member.

`Unit`

A PEP_PERF_STATE_UNIT structure that specifies the units in which the performance values for this P-state set are expressed. Component performance can be expressed in hertz (frequency) or in bits per second (bandwidth).

`Type`

A PEP_PERF_STATE_TYPE enumeration value that indicates the type of performance information that is specified for this component. This member indicates whether the performance values for this P-state set are specified as a list of discrete values or as a continuous range of values.

`Discrete`

`Discrete.Count`

`Discrete.States`

`Range`

`Range.Minimum`

`Range.Maximum`

Remarks

The **PerfStateSets** member of the PEP_COMPONENT_PERF_INFO is the first element in an array of **PEP_COMPONENT_PERF_SET** structures. All members of the **PEP_COMPONENT_PERF_SET** structure contain input values that are supplied by the Windows power management framework (PoFx). The platform extension plug-in (PEP) must not write to this structure.

Device drivers use the PO_FX_COMPONENT_PERF_SET structure, which is similar to the **PEP_COMPONENT_PERF_SET** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_PERF_STATE

PEP_PERF_STATE_TYPE

PEP_PERF_STATE_UNIT

PO_FX_COMPONENT_PERF_SET

UNICODE_STRING

PEP_COMPONENT_PERF_STATE_REQUEST structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_STATE_REQUEST** structure specifies a performance state (P-state) set and a new performance level to assign to this set.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_STATE_REQUEST {
  ULONG Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PEP_COMPONENT_PERF_STATE_REQUEST, *PPEP_COMPONENT_PERF_STATE_REQUEST;
```

Members

`Set`

The index of the P-state set to which to assign the new performance level. If N is the number of P-state sets specified for this component, P-state set indexes range from 0 to N−1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`StateIndex`

`StateValue`

Remarks

The **PerfRequests** member of the PEP_REQUEST_COMPONENT_PERF_STATE structure is a pointer to an array of **PEP_COMPONENT_PERF_STATE_REQUEST** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_REQUEST_COMPONENT_PERF_STATE

PEP_COMPONENT_PLATFORM_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PLATFORM_CONSTRAINTS** structure describes the lowest-powered F*x* state of that a component can be in when the platform is in a particular idle state.

Syntax

```
typedef struct _PEP_COMPONENT_PLATFORM_CONSTRAINTS {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  PULONG    MinimumFStates;
  ULONG     PlatformStateCount;
} PEP_COMPONENT_PLATFORM_CONSTRAINTS, *PPEP_COMPONENT_PLATFORM_CONSTRAINTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`MinimumFStates`

[in] A pointer to an output buffer. The PEP writes a ULONG array to this buffer that specifies the lowest-powered F*x* state that the component can be in for each platform idle state. An element with a value of 0 indicates F0, a value of 1 indicates F1, and so on. If the platform supports M idle states, array elements 0 to M−1 specify the F*x* states corresponding to platform idle states 0 to M−1.

`PlatformStateCount`

[in] The number of elements in the **MinimumFStates** array. The array contains one element for each platform idle state. The Windows power management framework (PoFx) previously sent a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification to the PEP to determine the number of supported platform idle states.

Remarks

This structure is used by the PEP_DPM_COMPONENT_IDLE_CONSTRAINTS notification. All four members of the structure contain input values that PoFx supplies when this notification is sent. PoFx allocates the buffer pointed to by the **MinimumFStates** member, and the PEP writes to this buffer in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_COMPONENT_IDLE_CONSTRAINTS

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_COMPONENT_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_V2** structure specifies the power state attributes of a component in the device.

Syntax

```
typedef struct _PEP_COMPONENT_V2 {
  GUID                          Id;
  ULONGLONG                     Flags;
  ULONG                         DeepestWakeableIdleState;
  ULONG                         IdleStateCount;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
} PEP_COMPONENT_V2, *PPEP_COMPONENT_V2;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The PEP should specify a nonzero value for this member if the Windows power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`Flags`

A set of component-power-state flags. No flags are currently defined for this member, which is always zero.

`DeepestWakeableIdleState`

The index of the deepest F*x* state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of F*x* power states that the component supports. A component must support at least one F*x* state (F0).

`IdleStates`

A pointer to an array of PO_FX_COMPONENT_IDLE_STATE structures. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an F*x* power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

Remarks

This structure

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PO_FX_COMPONENT_IDLE_STATE

PEP_COORDINATED_DEPENDENCY_OPTION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COORIDNATED_DEPENDENCY_OPTION** structure describes a coordinated idle state's dependency to the OS.

Syntax

```
typedef struct _PEP_COORDINATED_DEPENDENCY_OPTION {
  UCHAR   ExpectedStateIndex;
  BOOLEAN LooseDependency;
  BOOLEAN InitiatingState;
  BOOLEAN DependentState;
} PEP_COORDINATED_DEPENDENCY_OPTION, *PPEP_COORDINATED_DEPENDENCY_OPTION;
```

Members

`ExpectedStateIndex`

Supplies the index of the idle state (coordinated idle state or processor idle state) that must have already been entered/be entered by this idle transition.

`LooseDependency`

When **FALSE**, the OS must use synchronization to guarantee that the state referred to by this dependency has already been entered. When **TRUE**, the OS may perform best-effort coordination that the target state is entered. The PEP must set **LooseDependency** to **TRUE** if the **ExpectedState** is a processor state and has the **WakesSpuriously** flag set.

`InitiatingState`

Indicates if the OS may simultaneously enter the state referred to by this dependency. If **FALSE**, this state is only valid when another processor has previously entered it.

`DependentState`

Indicates if this state is a valid state for a dependent processor to be in.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY notification

PEP_COORDINATED_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COORIDNATED_IDLE_STATE** structure describes a coordinated idle state to the OS.

Syntax

```
typedef struct _PEP_COORDINATED_IDLE_STATE {
  ULONG Latency;
  ULONG BreakEvenDuration;
  ULONG DependencyCount;
  ULONG MaximumDependencySize;
} PEP_COORDINATED_IDLE_STATE, *PPEP_COORDINATED_IDLE_STATE;
```

Members

`Latency`

The latency of waking from this idle state, in 100ns units.

`BreakEvenDuration`

Supplies the minimum time the state must be entered to amortize the cost of entering/exiting the state. Idle durations longer than this period should save power when compared to entering a lighter state for the same period.

`DependencyCount`

Supplies the number of dependencies this coordinated state has on other coordinated states or on processors.

`MaximumDependencySize`

Supplies the maximum size of a single dependency.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COORDINATED_IDLE_STATE structure

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification

PEP_CRASHDUMP_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_CRASHDUMP_INFORMATION** structure contains information about a crash-dump device.

Syntax

```
typedef struct _PEP_CRASHDUMP_INFORMATION {
  PEPHANDLE DeviceHandle;
  PVOID     DeviceContext;
} PEP_CRASHDUMP_INFORMATION, *PPEP_CRASHDUMP_INFORMATION;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the crash-dump device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`DeviceContext`

[in] A pointer to a device-specific context. This pointer is the *Context* parameter value that the device driver previously passed to the PoFxPowerOnCrashdumpDevice routine. The context contains information in a format that is defined by the device driver and is understood by the PEP. The context is opaque to the operating system. This member value can be NULL if the PEP does not require a context.

Remarks

The *CrashdumpInformation* parameter to the PowerOnDumpDeviceCallback routine is a pointer to a **PEP_CRASHDUMP_INFORMATION** structure.

When a fatal system error occurs, the devices in the crash-dump device chain (storage controller, PCI controller, and so on) need to be turned on so that the Windows kernel can write a crash-dump file to disk.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PoFxPowerOnCrashdumpDevice

PowerOnDumpDeviceCallback

PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEBUGGER_TRANSITION_REQUIREMENTS** structure indicates the platform idle states for which the debugger device must be turned on.

Syntax

```
typedef struct _PEP_DEBUGGER_TRANSITION_REQUIREMENTS {
  PEPHANDLE DeviceHandle;
  ULONG     PlatformStateCount;
  PBOOLEAN  PowerTransitionRequired;
} PEP_DEBUGGER_TRANSITION_REQUIREMENTS, *PPEP_DEBUGGER_TRANSITION_REQUIREMENTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the debugger device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PlatformStateCount`

[in] The number of entries in the array pointed to by the **PowerTransitionRequired** member. This number is the same as the number of supported platform idle states that the PEP specified in response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

`PowerTransitionRequired`

[in] An output buffer to which the PEP writes a BOOLEAN array that indicates the platform idle states for which power to the debugger device must be turned on. This buffer is allocated by the Windows power management framework (PoFx) and is guaranteed to be large enough to contain the number of array elements specified by the **PlatformStateCount** member. Set an array element to TRUE if the debugger device must be turned on in the corresponding platform idle state. Set to FALSE to indicate that the device is not required to be turned on in this state. If the platform supports N idle states, the array contains N elements, and idle states are numbered 0 to N–1 in the order in which they are represented in the array.

Remarks

This structure is used by the PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS notification. All three members of the structure contain input values that are supplied by PoFx when this notification is sent to the PEP.

During a debugging session, the debugger component that runs on the target computer communicates with the host computer through a debugger device such as a USB host controller, UART, or networking device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_DEVICE_ACCEPTANCE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_ACCEPTANCE_TYPE** enumeration indicates whether a PEP accepts ownership of a device.

Syntax

```
typedef enum _PEP_DEVICE_ACCEPTANCE_TYPE {
  PepDeviceNotAccepted,
  PepDeviceAccepted,
  PepDeviceAceptedMax
} PEP_DEVICE_ACCEPTANCE_TYPE, *PPEP_DEVICE_ACCEPTANCE_TYPE;
```

Constants

PepDeviceNotAccepted	The PEP does not claim ownership of this device.
PepDeviceAccepted	The PEP claims ownership of this device.
PepDeviceAceptedMax	

Remarks

This enumeration is used by **DeviceAccepted** member of the PEP_REGISTER_DEVICE_V2 structure to indicate whether a PEP accepts ownership of a device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_REGISTER_DEVICE_V2

PEP_DEVICE_PLATFORM_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_PLATFORM_CONSTRAINTS** structure specifies the constraints for entry to the various D*x* power states that are supported by a device.

Syntax

```
typedef struct _PEP_DEVICE_PLATFORM_CONSTRAINTS {
  PEPHANDLE           DeviceHandle;
  PDEVICE_POWER_STATE MinimumDStates;
  ULONG               PlatformStateCount;
} PEP_DEVICE_PLATFORM_CONSTRAINTS, *PPEP_DEVICE_PLATFORM_CONSTRAINTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`MinimumDStates`

[in] A pointer to an array of DEVICE_POWER_STATE enumeration values that indicate the lowest-powered D*x* (device power) state the device can enter for each platform idle state.

`PlatformStateCount`

[in] The number of elements in the array pointed to by the **MinimumDStates** member. This member contains the platform state count that the PEP supplied in response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

Remarks

This structure is used by the PEP_DPM_DEVICE_IDLE_CONSTRAINTS notification. All three members of this structure contain input values that are supplied by the Windows power management framework (PoFx). In response to this notification, the PEP writes DEVICE_POWER_STATE enumeration values to the elements of the array pointed to by the **MinimumDStates** member. PoFx allocates the storage for this array before sending the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

DEVICE_POWER_STATE

PEP_DPM_DEVICE_IDLE_CONSTRAINTS

PEP_DPM_REGISTER_DEVICE

PEP_DEVICE_POWER_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_POWER_STATE** structure indicates the status of a transition to a new D*x* (device power) state.

Syntax

```
typedef struct _PEP_DEVICE_POWER_STATE {
  PEPHANDLE          DeviceHandle;
  DEVICE_POWER_STATE PowerState;
  BOOLEAN            Complete;
  BOOLEAN            SystemTransition;
} PEP_DEVICE_POWER_STATE, *PPEP_DEVICE_POWER_STATE;
```

Members

`DeviceHandle`

[in] The PEPHANDLE value that identifies this device. The PEP previously created this handle in response to a PEP_DPM_REGISTER_DEVICE notification from the Windows power management framework (PoFx).

`PowerState`

[in] A DEVICE_POWER_STATE enumeration value that specifies the new device power state.

`Complete`

[in] Whether the transition to the new device power state has just been initiated or has just completed. If TRUE, the transition to the target device power state has completed. If FALSE, the power policy owner (PPO) has initiated the transition by calling the PoRequestPowerIrp routine, but the Windows power manager has not yet issued the D*x* IRP (an IRP_MN_SET_POWER request of type **DevicePowerState**) to the device's driver stack.

`SystemTransition`

[in] Always set to FALSE.

Remarks

This structure is used by the PEP_DPM_DEVICE_POWER_STATE notification. All four members of the structure contain input values that are supplied by PoFx. The PEP does not write to this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

DEVICE_POWER_STATE

IRP_MN_SET_POWER

PEP_DPM_DEVICE_POWER_STATE

PEP_DPM_REGISTER_DEVICE

PoRequestPowerIrp

PEP_DEVICE_REGISTER_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_REGISTER** structure describes all the components in a particular device.

Syntax

```
typedef struct _PEP_DEVICE_REGISTER_V2 {
  ULONGLONG         Flags;
  ULONG             ComponentCount;
  PPEP_COMPONENT_V2 Components[ANYSIZE_ARRAY];
} PEP_DEVICE_REGISTER_V2, *PPEP_DEVICE_REGISTER_V2;
```

Members

`Flags`

TBD

`ComponentCount`

The number of components in this device, which is also the number of elements in the **Components** array. The **ComponentCount** value must be greater than or equal to one.

`Components`

The first element in an array of pointers to PEP_COMPONENT_V2 structures. Each element in the array points to a structure that describes a component in the device. If this array contains more than one element, the additional elements immediately follow the end of the **PEP_DEVICE_REGISTER** structure.

Remarks

The PEP_REGISTER_DEVICE structure contains a **Register** member that points to a **PEP_DEVICE_REGISTER** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_V2

PEP_REGISTER_DEVICE

PEP_DEVICE_STARTED structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_STARTED** structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_DEVICE_STARTED {
  PEPHANDLE DeviceHandle;
} PEP_DEVICE_STARTED, *PPEP_DEVICE_STARTED;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_DEVICE_STARTED notification. The **DeviceHandle** member of this structure contains an input value that is supplied by PoFx.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_DEVICE_STARTED

PEP_DPM_REGISTER_DEVICE

PEP_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_INFORMATION** structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_INFORMATION {
  USHORT                 Version;
  USHORT                 Size;
  PPEPCALLBACKNOTIFYDPM  AcceptDeviceNotification;
  PPEPCALLBACKNOTIFYPPM  AcceptProcessorNotification;
  PPEPCALLBACKNOTIFYACPI AcceptAcpiNotification;
} PEP_INFORMATION, *PPEP_INFORMATION;
```

Members

`Version`

The current version number for this structure. Set this member to PEP_INFORMATION_VERSION.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_INFORMATION**).

`AcceptDeviceNotification`

A pointer to an AcceptDeviceNotification callback routine. This member is required to point to a valid callback routine.

`AcceptProcessorNotification`

A pointer to an AcceptProcessorNotification callback routine. This member is optional and can be NULL if the PEP is not prepared to handle PPM notifications from PoFx.

`AcceptAcpiNotification`

A pointer to an AcceptAcpiNotification callback routine. This member is optional and can be NULL if the PEP is not prepared to handle ACPI notifications from PoFx.

Remarks

This structure contains pointers to several callback routines that are implemented by the PEP. PoFx calls these routines to send notifications to the PEP.

The *PepInformation* parameter to the PoFxRegisterPlugin and PoFxRegisterPluginEx routines is a pointer to a **PEP_INFORMATION** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

AcceptAcpiNotification

AcceptDeviceNotification

AcceptProcessorNotification

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEP_KERNEL_INFORMATION_STRUCT_V1 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_KERNEL_INFORMATION** structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_KERNEL_INFORMATION_STRUCT_V1 {
  USHORT                                  Version;
  USHORT                                  Size;
  POHANDLE                                Plugin;
  PPOFXCALLBACKREQUESTWORKER              RequestWorker;
  PPOFXCALLBACKENUMERATEUNMASKEDINTERRUPTS EnumerateUnmaskedInterrupts;
  PPOFXCALLBACKPROCESSORHALT              ProcessorHalt;
  PPOFXCALLBACKREQUESTINTERRUPT           RequestInterrupt;
  PPOFXCALLBACKCRITICALRESOURCE           TransitionCriticalResource;
} PEP_KERNEL_INFORMATION_STRUCT_V1, *PPEP_KERNEL_INFORMATION_STRUCT_V1;
```

Members

`Version`

The current version number for this structure. Set this member to PEP_KERNEL_INFORMATION_VERSION.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_KERNEL_INFORMATION**).

`Plugin`

The handle assigned to the PEP's registration with PoFx. PoFx sets the value of this member. The PEP uses this handle in calls to the RequestWorker routine.

`RequestWorker`

A pointer to a RequestWorker routine. PoFx sets the value of this member. The **RequestWorker** routine is implemented by PoFx. The PEP calls this routine to request the use of a worker thread from the operating system.

`EnumerateUnmaskedInterrupts`

A pointer to an EnumerateUnmaskedInterrupts routine. PoFx sets the value of this member. The **EnumerateUnmaskedInterrupts** routine is implemented by PoFx. The PEP calls this routine to request information about the unmasked interrupts.

`ProcessorHalt`

A pointer to a ProcessorHalt routine. PoFx sets the value of this member. The **ProcessorHalt** routine is implemented by PoFx. The PEP calls this routine to prepare the current processor to enter the halted state.

`RequestInterrupt`

A pointer to a RequestInterrupt routine. PoFx sets the value of this member. The **RequestInterrupt** routine is implemented by PoFx. The PEP calls this routine to replay an edge-triggered interrupt that might have been lost after the hardware platform entered a low-power system state.

`TransitionCriticalResource`

A pointer to a TransitionCriticalResource routine. PoFx sets the value of this member. The **TransitionCriticalResource** routine is implemented by PoFx. The PEP calls this routine to transition critical system resources to the idle condition.

Remarks

The **KernelInformation** parameter to the PoFxRegisterPlugin or PoFxRegisterPluginEx routine is a pointer to a **PEP_KERNEL_INFORMATION** structure. The PEP allocates this structure and sets the values of the **Version** and **Size** members of this structure before calling **PoFxRegisterPlugin** or **PoFxRegisterPluginEx**. During the call to this routine, PoFx fills in the remaining members of the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h

See Also

CompleteWork

EnumerateUnmaskedInterrupts

PlatformIdleVeto

PoFxRegisterPlugin

PoFxRegisterPluginEx

ProcessorHalt

ProcessorIdleVeto

RequestCommon

RequestInterrupt

RequestWorker

TransitionCriticalResource

UpdatePlatformIdleState

UpdateProcessorIdleState

PEP_KERNEL_INFORMATION_STRUCT_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_KERNEL_INFORMATION** structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_KERNEL_INFORMATION_STRUCT_V2 {
  USHORT                                  Version;
  USHORT                                  Size;
  POHANDLE                                Plugin;
  PPOFXCALLBACKREQUESTWORKER              RequestWorker;
  PPOFXCALLBACKENUMERATEUNMASKEDINTERRUPTS EnumerateUnmaskedInterrupts;
  PPOFXCALLBACKPROCESSORHALT              ProcessorHalt;
  PPOFXCALLBACKREQUESTINTERRUPT           RequestInterrupt;
  PPOFXCALLBACKCRITICALRESOURCE           TransitionCriticalResource;
  PPOFXCALLBACKPROCESSORIDLEVETO          ProcessorIdleVeto;
  PPOFXCALLBACKPLATFORMIDLEVETO           PlatformIdleVeto;
  PPOFXCALLBACKUPDATEPROCESSORIDLESTATE   UpdateProcessorIdleState;
  PPOFXCALLBACKUPDATEPLATFORMIDLESTATE    UpdatePlatformIdleState;
} PEP_KERNEL_INFORMATION_STRUCT_V2, *PPEP_KERNEL_INFORMATION_STRUCT_V2;
```

Members

`Version`

The current version number for this structure. Set this member to PEP_KERNEL_INFORMATION_VERSION.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_KERNEL_INFORMATION**).

`Plugin`

The handle assigned to the PEP's registration with PoFx. PoFx sets the value of this member. The PEP uses this handle in calls to the RequestWorker routine.

`RequestWorker`

A pointer to a RequestWorker routine. PoFx sets the value of this member. The **RequestWorker** routine is implemented by PoFx. The PEP calls this routine to request the use of a worker thread from the operating system.

`EnumerateUnmaskedInterrupts`

A pointer to an EnumerateUnmaskedInterrupts routine. PoFx sets the value of this member. The **EnumerateUnmaskedInterrupts** routine is implemented by PoFx. The PEP calls this routine to request information about the unmasked interrupts.

`ProcessorHalt`

A pointer to a ProcessorHalt routine. PoFx sets the value of this member. The **ProcessorHalt** routine is implemented by PoFx. The PEP calls this routine to prepare the current processor to enter the halted state.

`RequestInterrupt`

A pointer to a RequestInterrupt routine. PoFx sets the value of this member. The **RequestInterrupt** routine is implemented by PoFx. The PEP calls this routine to replay an edge-triggered interrupt that might have been lost after the hardware platform entered a low-power system state.

`TransitionCriticalResource`

A pointer to a TransitionCriticalResource routine. PoFx sets the value of this member. The **TransitionCriticalResource** routine is implemented by PoFx. The PEP calls this routine to transition critical system resources to the idle condition.

`ProcessorIdleVeto`

A pointer to a ProcessorIdleVeto routine. PoFx sets the value of this member. The **ProcessorIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a processor idle state.

`PlatformIdleVeto`

A pointer to a PlatformIdleVeto routine. PoFx sets the value of this member. The **PlatformIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a platform idle state.

`UpdateProcessorIdleState`

A pointer to an UpdateProcessorIdleState routine. PoFx sets the value of this member. The **UpdateProcessorIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified processor idle state.

`UpdatePlatformIdleState`

A pointer to an UpdatePlatformIdleState routine. PoFx sets the value of this member. The **UpdatePlatformIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified platform idle state.

Remarks

The **KernelInformation** parameter to the PoFxRegisterPlugin or PoFxRegisterPluginEx routine is a pointer to a **PEP_KERNEL_INFORMATION** structure. The PEP allocates this structure and sets the values of the **Version** and **Size** members of this structure before calling **PoFxRegisterPlugin** or **PoFxRegisterPluginEx**. During the call to this routine, PoFx fills in the remaining members of the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h

See Also

CompleteWork

EnumerateUnmaskedInterrupts

PlatformIdleVeto

PoFxRegisterPlugin

PoFxRegisterPluginEx

ProcessorHalt

ProcessorIdleVeto

RequestCommon

RequestInterrupt

RequestWorker

TransitionCriticalResource

UpdatePlatformIdleState

UpdateProcessorIdleState

PEP_KERNEL_INFORMATION_STRUCT_V3 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_KERNEL_INFORMATION** structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_KERNEL_INFORMATION_STRUCT_V3 {
  USHORT                                  Version;
  USHORT                                  Size;
  POHANDLE                                Plugin;
  PPOFXCALLBACKREQUESTWORKER              RequestWorker;
  PPOFXCALLBACKENUMERATEUNMASKEDINTERRUPTS EnumerateUnmaskedInterrupts;
  PPOFXCALLBACKPROCESSORHALT              ProcessorHalt;
  PPOFXCALLBACKREQUESTINTERRUPT           RequestInterrupt;
  PPOFXCALLBACKCRITICALRESOURCE           TransitionCriticalResource;
  PPOFXCALLBACKPROCESSORIDLEVETO          ProcessorIdleVeto;
  PPOFXCALLBACKPLATFORMIDLEVETO           PlatformIdleVeto;
  PPOFXCALLBACKUPDATEPROCESSORIDLESTATE   UpdateProcessorIdleState;
  PPOFXCALLBACKUPDATEPLATFORMIDLESTATE    UpdatePlatformIdleState;
  PPOFXCALLBACKREQUESTCOMMON              RequestCommon;
} PEP_KERNEL_INFORMATION_STRUCT_V3, *PPEP_KERNEL_INFORMATION_STRUCT_V3;
```

Members

`Version`

The current version number for this structure. Set this member to PEP_KERNEL_INFORMATION_VERSION.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_KERNEL_INFORMATION**).

`Plugin`

The handle assigned to the PEP's registration with PoFx. PoFx sets the value of this member. The PEP uses this handle in calls to the RequestWorker routine.

`RequestWorker`

A pointer to a RequestWorker routine. PoFx sets the value of this member. The **RequestWorker** routine is implemented by PoFx. The PEP calls this routine to request the use of a worker thread from the operating system.

`EnumerateUnmaskedInterrupts`

A pointer to an EnumerateUnmaskedInterrupts routine. PoFx sets the value of this member. The **EnumerateUnmaskedInterrupts** routine is implemented by PoFx. The PEP calls this routine to request information about the unmasked interrupts.

`ProcessorHalt`

A pointer to a ProcessorHalt routine. PoFx sets the value of this member. The **ProcessorHalt** routine is implemented by PoFx. The PEP calls this routine to prepare the current processor to enter the halted state.

`RequestInterrupt`

A pointer to a RequestInterrupt routine. PoFx sets the value of this member. The **RequestInterrupt** routine is implemented by PoFx. The PEP calls this routine to replay an edge-triggered interrupt that might have been lost after the hardware platform entered a low-power system state.

`TransitionCriticalResource`

A pointer to a TransitionCriticalResource routine. PoFx sets the value of this member. The **TransitionCriticalResource** routine is implemented by PoFx. The PEP calls this routine to transition critical system resources to the idle condition.

`ProcessorIdleVeto`

A pointer to a ProcessorIdleVeto routine. PoFx sets the value of this member. The **ProcessorIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a processor idle state.

`PlatformIdleVeto`

A pointer to a PlatformIdleVeto routine. PoFx sets the value of this member. The **PlatformIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a platform idle state.

`UpdateProcessorIdleState`

A pointer to an UpdateProcessorIdleState routine. PoFx sets the value of this member. The **UpdateProcessorIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified processor idle state.

`UpdatePlatformIdleState`

A pointer to an UpdatePlatformIdleState routine. PoFx sets the value of this member. The **UpdatePlatformIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified platform idle state.

`RequestCommon`

A pointer to an RequestCommon routine. PoFx sets the value of this member. The **RequestCommon** routine is implemented by PoFx. The PEP calls this routine to perform a processing operation that is specified by a request ID.

Remarks

The **KernelInformation** parameter to the PoFxRegisterPlugin or PoFxRegisterPluginEx routine is a pointer to a **PEP_KERNEL_INFORMATION** structure. The PEP allocates this structure and sets the values of the **Version** and **Size** members of this structure before calling **PoFxRegisterPlugin** or **PoFxRegisterPluginEx**. During the call to this routine, PoFx fills in the remaining members of the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h

See Also

CompleteWork

EnumerateUnmaskedInterrupts

PlatformIdleVeto

PoFxRegisterPlugin

PoFxRegisterPluginEx

ProcessorHalt

ProcessorIdleVeto

RequestCommon

RequestInterrupt

RequestWorker

TransitionCriticalResource

UpdatePlatformIdleState

UpdateProcessorIdleState

PEP_LOW_POWER_EPOCH structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_LOW_POWER_EPOCH** structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).

Syntax

```
typedef struct _PEP_LOW_POWER_EPOCH {
  BOOLEAN LowPowerEpoch;
} PEP_LOW_POWER_EPOCH, *PPEP_LOW_POWER_EPOCH;
```

Members

`LowPowerEpoch`

When **TRUE**, indicates that the low power epoch is on.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_LOW_POWER_EPOCH notification

PEP_NOTIFY_COMPONENT_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_NOTIFY_COMPONENT_IDLE_STATE** structure contains status information about a component's pending transition to a new F*x* power state.

Syntax

```
typedef struct _PEP_NOTIFY_COMPONENT_IDLE_STATE {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     IdleState;
  BOOLEAN   DriverNotified;
  BOOLEAN   Completed;
} PEP_NOTIFY_COMPONENT_IDLE_STATE, *PPEP_NOTIFY_COMPONENT_IDLE_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`IdleState`

[in] The new F*x* power state that the component will transition to. If this parameter is zero, the new state is F0; if this parameter is one, the new state is F1; and so on.

`DriverNotified`

[in] Whether the driver for the device has already been notified of the pending transition to the new F*x* state. If TRUE, the power management framework (PoFx) already notified the driver by calling the driver's ComponentIdleStateCallback callback routine. If FALSE, PoFx has not yet notified the driver.

`Completed`

[out] Whether the PEP has completed all necessary preparations for the specified component to enter the new F*x* state. Set to TRUE to indicate that the PEP has completed all necessary preparations, or to FALSE to indicate that is has not.

If FALSE, the PEP must complete the necessary preparations at a later time and then inform PoFx when the preparations have been completed. To do so, the PEP calls the RequestWorker routine to request a work item, and then responds to the ensuing PEP_DPM_WORK notification by submitting a work request of type **PepWorkCompleteIdleState** to inform PoFx that the preparations are complete.

Remarks

This structure is used by the PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE notification. The first four members of this structure contain input values that are supplied by PoFx. The **Completed** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE

PEP_DPM_REGISTER_DEVICE

PEP_DPM_WORK

RequestWorker

PEP_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE** structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.

Syntax

```
typedef struct _PEP_PERF_STATE {
  ULONGLONG Value;
  PVOID     Context;
} PEP_PERF_STATE, *PPEP_PERF_STATE;
```

Members

`Value`

The discrete value represented by this P-state. For more information, see Remarks.

`Context`

A pointer to PEP-defined context data. The PEP uses this context to contain additional information about the discrete performance level that cannot be expressed in the **Value** member. This context is opaque to the Windows power management framework (PoFx). The **Context** member is optional and can be set to NULL.

Remarks

The **Discrete.States** member of the PEP_COMPONENT_PERF_SET structure is a pointer to an array of **PEP_PERF_STATE** structures. The **Unit** member of the **PEP_COMPONENT_PERF_SET** structure specifies the units in which the **Value** member in each array element is expressed. Component performance can be expressed in hertz (frequency) or in bits per second (bandwidth). For example, if **Value** = 100,000,000 and **Unit** = **PepPerfStateUnitFrequency**, this performance state represents a frequency of 100 megahertz.

Device drivers use the PO_FX_PERF_STATE structure, which is similar to the **PEP_PERF_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PO_FX_PERF_STATE

PEP_PERF_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE_TYPE** enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.

Syntax

```
typedef enum _PEP_PERF_STATE_TYPE {
  PepPerfStateTypeDiscrete,
  PepPerfStateTypeRange,
  PepPerfStateTypeMax
} PEP_PERF_STATE_TYPE, *PPEP_PERF_STATE_TYPE;
```

Constants

PepPerfStateTypeDiscrete	The performance information is specified as a set of discrete values.
PepPerfStateTypeRange	The performance information is specified as a range of values.
PepPerfStateTypeMax	Reserved for use by operating system.

Remarks

The **Type** member of the PEP_COMPONENT_PERF_SET structure is a **PEP_PERF_STATE_TYPE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PEP_PERF_STATE_UNIT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE_UNIT** enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.

Syntax

```
typedef enum _PEP_PERF_STATE_UNIT {
  PepPerfStateUnitOther,
  PepPerfStateUnitFrequency,
  PepPerfStateUnitBandwidth,
  PepPerfStateUnitMax
} PEP_PERF_STATE_UNIT, *PPEP_PERF_STATE_UNIT;
```

Constants

PepPerfStateUnitOther	Some other type of measurement units not explicitly described by this enumeration.
PepPerfStateUnitFrequency	Hertz (frequency units).
PepPerfStateUnitBandwidth	Bits per second (bandwidth units).
PepPerfStateUnitMax	Reserved for use by operating system.

Remarks

The **Type** member of the PEP_COMPONENT_PERF_SET structure is a **PEP_PERF_STATE_UNIT** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PEP_PLATFORM_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PLATFORM_IDLE_STATE** structure specifies the properties of a platform idle state.

Syntax

```
typedef struct _PEP_PLATFORM_IDLE_STATE {
  POHANDLE                      InitiatingProcessor;
  UCHAR                         InitiatingState;
  ULONG                         Latency;
  ULONG                         BreakEvenDuration;
  ULONG                         DependencyArrayUsed;
  ULONG                         DependencyArrayCount;
  PEP_PROCESSOR_IDLE_DEPENDENCY DependencyArray[ANYSIZE_ARRAY];
} PEP_PLATFORM_IDLE_STATE, *PPEP_PLATFORM_IDLE_STATE;
```

Members

`InitiatingProcessor`

A **POHANDLE** value that identifies the processor that initiates the transition to this platform idle state, or **NULL** if any processor can initiate the transition. If non-NULL, this handle represents the registration of the processor (as a device) with the Windows power management framework (PoFx).

`InitiatingState`

The index of the processor idle state that the processor enters to initiate the platform's entry to the specified platform idle state. If the **IdleStates** array in the PEP_PPM_QUERY_IDLE_STATES_V2 structure contains N elements, the idle states are numbered 0 to N-1 in the order in which they appear in the array.

`Latency`

The worst-case latency, in 100-nanosecond units, that the platform requires to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. PoFx uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

`DependencyArrayUsed`

The number of items in **DependencyArray** which were filled in by the PEP.

`DependencyArrayCount`

The number of elements in the **DependencyArray** array. The array contains one element for each processor in the hardware platform.

`DependencyArray`

The first element in an array of PEP_PROCESSOR_IDLE_DEPENDENCY structures. This array specifies the set of

dependencies that the platform idle state has on each processor. If the platform contains N processors, the array contains N elements, and processors are numbered 0 to N-1 in the order in which they are represented in the array.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification. The **State** member of a PEP_PPM_QUERY_PLATFORM_STATE structure is a **PEP_PLATFORM_IDLE_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PPM_QUERY_PLATFORM_STATE

PEP_PROCESSOR_IDLE_DEPENDENCY

PEP_PLATFORM_IDLE_STATE_UPDATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PLATFORM_IDLE_STATE_UPDATE** structure contains the updated properties of a platform idle state.

Syntax

```
typedef struct _PEP_PLATFORM_IDLE_STATE_UPDATE {
  ULONG Version;
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PLATFORM_IDLE_STATE_UPDATE, *PPEP_PLATFORM_IDLE_STATE_UPDATE;
```

Members

`Version`

The current version number of this structure. Set this member to PEP_PLATFORM_IDLE_STATE_UPDATE_VERSION.

`Latency`

The worst-case latency, in 100-nanosecond units, for the platform to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. The Windows power management framework (PoFx) uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

The *Update* parameter of the UpdatePlatformIdleState routine is a pointer to a **PEP_PLATFORM_IDLE_STATE_UPDATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

UpdatePlatformIdleState

PEP_POWER_CONTROL_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_POWER_CONTROL_COMPLETE** structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.

Syntax

```
typedef struct _PEP_POWER_CONTROL_COMPLETE {
  PEPHANDLE DeviceHandle;
  LPCGUID   PowerControlCode;
  PVOID     RequestContext;
  SIZE_T    BytesReturned;
  NTSTATUS  Status;
} PEP_POWER_CONTROL_COMPLETE, *PPEP_POWER_CONTROL_COMPLETE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation that was performed. This is the same value that the PEP supplied in response to the PEP_DPM_WORK notification to initiate the power control operation.

`RequestContext`

[in] A pointer to the request context that was sent by the PEP in the PEP_WORK_INFORMATION structure that the PEP supplied in the original work request. Typically, this member points to a structure that contains a pointer to an output buffer to contain the results of the power control operation that was requested by the PEP.

`BytesReturned`

[in] The size, in bytes, of the result data stored by the driver in the output buffer. For more information about this buffer, see the description of the **RequestContext** member.

`Status`

[in] The status of the power control operation. If the operation was successful, the PEP sets this member to STATUS_SUCCESS. Otherwise, the PEP sets this member to an appropriate error status code.

Remarks

This structure is used by the PEP_DPM_POWER_CONTROL_COMPLETE notification. All five members of the structure contain input values that are supplied by

If the output buffer is too small to receive all of the result data from the operation, the PEP sets the **Status** member of the structure to STATUS_INSUFFICIENT_RESOURCES, sets the **BytesReturned** member to the required size of the output buffer, and (typically) writes no data to the output buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

GUID

PEP_POWER_CONTROL_REQUEST structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_POWER_CONTROL_REQUEST** structure contains a request from a driver for a power control operation.

Syntax

```
typedef struct _PEP_POWER_CONTROL_REQUEST {
  PEPHANDLE DeviceHandle;
  LPCGUID   PowerControlCode;
  PVOID     InBuffer;
  SIZE_T    InBufferSize;
  PVOID     OutBuffer;
  SIZE_T    OutBufferSize;
  SIZE_T    BytesReturned;
  NTSTATUS  Status;
} PEP_POWER_CONTROL_REQUEST, *PPEP_POWER_CONTROL_REQUEST;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation to perform. This is the same value that the requesting driver supplied as the *PowerControlCode* parameter to the PoFxPowerControl routine.

`InBuffer`

[in] A pointer to a driver-allocated input buffer that contains the input parameters for this power control operation. This is the same value that the requesting driver supplied as the *InBuffer* parameter to the **PoFxPowerControl** routine.

`InBufferSize`

[in] The size in bytes of the buffer pointed to by **InBuffer**. This is the same value that the requesting driver supplied as the *InBufferSize* parameter to the **PoFxPowerControl** routine.

`OutBuffer`

[in] A pointer to a driver-allocated output buffer to which the PEP writes the results of this power control operation. This is the same value that the requesting driver supplied as the *OutBuffer* parameter to the **PoFxPowerControl** routine.

`OutBufferSize`

[in] The size in bytes of the buffer pointed to by **OutBuffer**.

`BytesReturned`

[out] The number of bytes that the PEP wrote to the output buffer pointed to by **OutBuffer**.

`Status`

[out] The status of the requested power control operation. If the operation is successful, the PEP sets this member to STATUS_SUCCESS. Otherwise, the PEP sets this member to an appropriate error status code.

Remarks

This structure is used by the PEP_DPM_POWER_CONTROL_REQUEST notification. The first six members of the structure contain input values that are supplied by the Windows power management Framework (PoFx). The last two members contain values that the PEP writes to the structure in response to this notification. In addition, the PEP writes to the output buffer pointed to by the **OutBuffer** member.

If the output buffer is too small to receive all of the result data from the operation, the PEP sets the **Status** member of the structure to STATUS_INSUFFICIENT_RESOURCES, sets the **BytesReturned** member to the required size of the output buffer, and (typically) writes no data to the output buffer.

The driver for a device can call the PoFxPowerControl routine to request a power control operation on the device. During this call, PoFx delegates the handling of this request to the PEP that owns the device. The **PowerControlCode** member of the **PEP_POWER_CONTROL_REQUEST** structure contains a custom control code whose meaning is understood by both the driver and the PEP for the device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_POWER_CONTROL_REQUEST

PEP_DPM_REGISTER_DEVICE

PoFxPowerControl

PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CONTEXT_QUERY_PARKING_PAGE** structure describes the parking page for a processor.

Syntax

```
typedef struct _PEP_PPM_CONTEXT_QUERY_PARKING_PAGE {
  PHYSICAL_ADDRESS PhysicalPageAddress;
  PVOID            VirtualPageAddress;
} PEP_PPM_CONTEXT_QUERY_PARKING_PAGE, *PPEP_PPM_CONTEXT_QUERY_PARKING_PAGE;
```

Members

`PhysicalPageAddress`

The physical memory address of the parking page.

`VirtualPageAddress`

The virtual memory address of the parking page.

Remarks

The output buffer for a PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE power control request is a **PEP_PPM_CONTEXT_QUERY_PARKING_PAGE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE

PEP_PPM_CST_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CST_STATE** structure specifies the properties of a C state (ACPI processor power state).

Syntax

```
typedef struct _PEP_PPM_CST_STATE {
  UCHAR            Type;
  USHORT           Latency;
  ULONG            Power;
  UCHAR            AddressSpaceId;
  UCHAR            BitWidth;
  UCHAR            BitOffset;
  UCHAR            AccessSize;
  PHYSICAL_ADDRESS Address;
} PEP_PPM_CST_STATE, *PPEP_PPM_CST_STATE;
```

Members

`Type`

The C-state type (0 = C0, 1 = C1, 2 = C2, and so on).

`Latency`

The worst-case latency, in microseconds, to enter and exit this C state. There are no restrictions on latency size.

`Power`

The average power consumption, in milliwatts, of the processor when in this C state.

`AddressSpaceId`

The register address space ID (_ASI) for this C state. This member specifies the address space in which the register for this C state is located. The following IDs are defined for this member.

ID	MEANING
0x00	System memory space
0x01	System I/O space
0x02	PCI configuration space
0x03	Embedded controller
0x04	SMBus address
0x0A	Platform communications channel (PCC)
0x7F	Functional fixed hardware

`BitWidth`

The register bit width (_RBW) for this C state. This member specifies the width, in bits, of the register for this C state.

`BitOffset`

The register bit offset (_RBO) for this C state. This member specifies the offset, in bits, from the register address specified in the **Address** member to the start of the register for this C state.

`AccessSize`

The register access size (_ASZ) code for this C state. This member specifies the number of bytes to read during an access of the register for this C state. Valid access size codes are shown in the following table.

ACCESS SIZE CODE	MEANING
0	Undefined
1	BYTE (1 byte)
2	WORD (2 bytes)
3	DWORD (4 bytes)
4	QWORD (8 bytes)

`Address`

The register address (_ADR) for this C state.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_CST_STATES notification. The **IdleStates** member of the PEP_PPM_CST_STATES structure is a pointer to an array of **PEP_PPM_CST_STATE** structures. The **Type** member in each element of the array should equal the array index of this member.

The values in the **PEP_PPM_CST_STATE** structure are obtained from the parameters that are specified for this C state in the _CST object in the ACPI namespace for the processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_CST_STATES

PEP_PPM_CST_STATES

PEP_PPM_CST_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CST_STATES** structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.

Syntax

```
typedef struct _PEP_PPM_CST_STATES {
  ULONG              Count;
  PPEP_PPM_CST_STATE IdleStates;
} PEP_PPM_CST_STATES, *PPEP_PPM_CST_STATES;
```

Members

`Count`

The number of elements in the **IdleStates** array.

`IdleStates`

A pointer to an array of PEP_PPM_CST_STATE structures.

Remarks

This structure is used by the PEP_NOTIFY_PPM_CST_STATES notification. The contents of this structure are obtained from the _CST object that is located in the ACPI namespace for the processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_CST_STATES

PEP_PPM_CST_STATE

PEP_PPM_ENTER_SYSTEM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE** notification to notify PEP that the system is about to enter a system power state.

Syntax

```
typedef struct _PEP_PPM_ENTER_SYSTEM_STATE {
  SYSTEM_POWER_STATE TargetState;
} PEP_PPM_ENTER_SYSTEM_STATE, *PPEP_PPM_ENTER_SYSTEM_STATE;
```

Members

`TargetState`

On input, a SYSTEM_POWER_STATE-type value that indicates the power state that the system is entering.

Remarks

This notification is sent to all processors simultaneously after the system has completed all passive level work transitioning the processor to the system power state. This notification is sent at DISPATCH_LEVEL, with all processors at dispatch. This notification is always executed on the target processor. PEP must not queue any work from this notification. The processors will not process work items, DPCs, etc. after this notification has been sent.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pep_x.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_FEEDBACK_READ structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_FEEDBACK_READ** structure contains the value read from a processor performance feedback counter.

Syntax

```
typedef struct _PEP_PPM_FEEDBACK_READ {
  ULONG CounterIndex;
  union {
    ULONG64 InstantaneousValue;
    struct {
      ULONG64 NominalCount;
      ULONG64 ActualCount;
    };
  };
} PEP_PPM_FEEDBACK_READ, *PPEP_PPM_FEEDBACK_READ;
```

Members

`CounterIndex`

[in] The index that identifies which processor performance feedback counter to read. If the platform extension plug-in (PEP) supports N counters for this processor, counter indexes range from 0 to N-1. The PEP previously supplied the number of supported counters in response to a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification.

`InstantaneousValue`

`NominalCount`

`ActualCount`

Remarks

This structure is used by the PEP_NOTIFY_PPM_FEEDBACK_READ notification. The **CounterIndex** member of the structure contains an input value supplied by the Windows power management framework (PoFx) when this notification is set. The other members contain output values that the PEP writes to the structure in response to the notification. The PEP writes to the **InstantaneousValue** member if the counter generates an instantaneous value, or to the **NominalCount** and **ActualCount** members if the counter generates a relative value.

Both an instantaneous counter and a relative counter are reset to zero when power is first turned on, but reading a relative counter causes the count to reset to zero, whereas reading an instantaneous counter does not reset the count. The PEP previously indicated whether the counter is instantaneous or relative in response to a PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_FEEDBACK_READ

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

PEP_PPM_IDLE_CANCEL structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_CANCEL** structure indicates why the processor could not enter the previously selected idle state.

Syntax

```
typedef struct _PEP_PPM_IDLE_CANCEL {
  PEP_PROCESSOR_IDLE_CANCEL_CODE CancelCode;
} PEP_PPM_IDLE_CANCEL, *PPEP_PPM_IDLE_CANCEL;
```

Members

`CancelCode`

[in] A PEP_PROCESSOR_IDLE_CANCEL_CODE enumeration value that indicates why the processor could not enter the idle state selected by the platform extension plug-in (PEP).

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_CANCEL notification. The **CancelCode** member of the structure contains an input value that the Windows power management framework (PoFx) supplies before this notification is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_NOTIFY_PPM_IDLE_CANCEL

PEP_PROCESSOR_IDLE_CANCEL_CODE

PEP_PPM_IDLE_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_COMPLETE** structure describe the idle states from which the processor and hardware platform are waking.

Syntax

```
typedef struct _PEP_PPM_IDLE_COMPLETE {
  ULONG ProcessorState;
  ULONG PlatformState;
} PEP_PPM_IDLE_COMPLETE, *PPEP_PPM_IDLE_COMPLETE;
```

Members

`ProcessorState`

[in] An index that identifies the idle state from which the processor is waking. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1. When the hypervisor is enabled and the platform wakes from an idle state, this member is set to **PEP_PROCESSOR_IDLE_STATE_UNKNOWN** (0xffffffff). For more information, see PEP_NOTIFY_PPM_IDLE_COMPLETE.

`PlatformState`

[in] An index that identifies the idle state from which the platform is waking. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If the transition from the processor idle state does not involve a transition from an platform idle state, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_COMPLETE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_COMPLETE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_IDLE_COMPLETE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_COMPLETE_V2** structure describe the idle states from which the processor and hardware platform are waking.

Syntax

```
typedef struct _PEP_PPM_IDLE_COMPLETE_V2 {
  ULONG  ProcessorState;
  ULONG  PlatformState;
  ULONG  CoordinatedStateCount;
  PULONG CoordinatedStates;
} PEP_PPM_IDLE_COMPLETE_V2, *PPEP_PPM_IDLE_COMPLETE_V2;
```

Members

`ProcessorState`

[in] An index that identifies the idle state from which the processor is waking. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1. When the hypervisor is enabled and the platform wakes from an idle state, this member is set to **PEP_PROCESSOR_IDLE_STATE_UNKNOWN** (0xffffffff). For more information, see PEP_NOTIFY_PPM_IDLE_COMPLETE.

`PlatformState`

[in] An index that identifies the idle state from which the platform is waking. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If the transition from the processor idle state does not involve a transition from an platform idle state, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`CoordinatedStateCount`

`CoordinatedStates`

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_COMPLETE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

PEP_PPM_IDLE_EXECUTE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_EXECUTE** structure specifies the idle state that the processor is to enter.

Syntax

```
typedef struct _PEP_PPM_IDLE_EXECUTE {
  NTSTATUS Status;
  ULONG    ProcessorState;
  ULONG    PlatformState;
} PEP_PPM_IDLE_EXECUTE, *PPEP_PPM_IDLE_EXECUTE;
```

Members

`Status`

[out] An **NTSTATUS** value that indicates whether the processor idle state transition was successful. The platform extension plug-in (PEP) sets this member to **STATUS_SUCCESSFUL** if the transition succeeded. Otherwise, this member is set to an appropriate error status code.

`ProcessorState`

[in] The index of the processor idle state that the processor is to enter. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value P**EP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_EXECUTE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The **Status** member contains an output value that the PEP writes to this member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_EXECUTE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_IDLE_EXECUTE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_EXECUTE_V2** structure specifies the idle state that the processor is to enter.

Syntax

```
typedef struct _PEP_PPM_IDLE_EXECUTE_V2 {
  NTSTATUS Status;
  ULONG    ProcessorState;
  ULONG    PlatformState;
  ULONG    CoordinatedStateCount;
  PULONG   CoordinatedStates;
} PEP_PPM_IDLE_EXECUTE_V2, *PPEP_PPM_IDLE_EXECUTE_V2;
```

Members

`Status`

[out] An **NTSTATUS** value that indicates whether the processor idle state transition was successful. The platform extension plug-in (PEP) sets this member to **STATUS_SUCCESSFUL** if the transition succeeded. Otherwise, this member is set to an appropriate error status code.

`ProcessorState`

[in] The index of the processor idle state that the processor is to enter. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`CoordinatedStateCount`

Supplies the number of coordinated idle states being entered by this transition.

`CoordinatedStates`

Supplies a pointer to an array of coordinated idle states that are being entered by this transition.

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_EXECUTE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The **Status** member contains an output value that the PEP writes to this member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_EXECUTE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_IDLE_SELECT structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_SELECT** structure describes the most energy-efficient idle state that the processor can enter and still satisfy the constraints specified by the operating system.

Syntax

```
typedef struct _PEP_PPM_IDLE_SELECT {
  PPEP_PROCESSOR_IDLE_CONSTRAINTS Constraints;
  BOOLEAN                         AbortTransition;
  ULONG                           IdleStateIndex;
  ULONG                           DependencyArrayUsed;
  ULONG                           DependencyArrayCount;
  PPEP_PROCESSOR_IDLE_DEPENDENCY  DependencyArray;
  ULONG                           PlatformIdleStateIndex;
} PEP_PPM_IDLE_SELECT, *PPEP_PPM_IDLE_SELECT;
```

Members

`Constraints`

[in] A pointer to a PEP_PROCESSOR_IDLE_CONSTRAINTS structure that specifies the constraints that the selected processor idle state must satisfy. The operating system sets the values in this structure. The platform extension plug-in (PEP) must select a processor idle state that satisfies these constraints.

`AbortTransition`

[out] Whether to cancel the pending transition to a processor idle state. Set this member to TRUE if the PEP requires the operating system to cancel the pending transition, and to FALSE otherwise. The PEP selects this option if it wants to queue work and have the Windows kernel process this work before the processor enters idle.

`IdleStateIndex`

[out] The index that identifies the idle state that the processor is to enter. This member is ignored if **AbortTransition** = TRUE. If the processor supports N idle states, idle state indexes range from 0 to N–1. The PEP supplied the number of supported idle states for this processor in response to a previous PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification.

`DependencyArrayUsed`

[out] The number of array elements that the PEP has actually written to the output buffer pointed to the **DependencyArray** member.

`DependencyArrayCount`

[in] The maximum number of elements in the array pointed to by the **DependencyArray** member. The **DependencyArrayCount** member value is equal to the **MaximumCoordinatedProcessors** value that the PEP previously supplied in response to the PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification.

`DependencyArray`

[in] A pointer to an output buffer to which the PEP writes an array of PEP_PROCESSOR_IDLE_DEPENDENCY structures. The elements of this array specify processor idle dependencies or wake dependencies that must be met

for this idle transition to succeed. The PEP sets the **DependencyArrayUsed** member to the actual number of array elements that the PEP writes to the buffer. The output buffer is allocated by the Windows power management framework (PoFx) and is guaranteed to be large enough to contain the number of array elements specified by the **DependencyArrayCount** member.

`PlatformIdleStateIndex`

[out] The index of the platform idle state that the platform is to enter when the processor enters the selected processor idle state. The PEP should set this member only if the platform enters an idle state at the same time as the processor. This member is ignored if **AbortTransition** is TRUE.

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_SELECT notification. The **Constraints**, **DependencyArrayCount**, and **DependencyArray** members of the structure contain input values supplied by PoFx when this notification is sent. The remaining members contain output values that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_NOTIFY_PPM_IDLE_SELECT

PEP_NOTIFY_PPM_QUERY_IDLE_STATES

PEP_PROCESSOR_IDLE_CONSTRAINTS

PEP_PROCESSOR_IDLE_DEPENDENCY

PEP_PPM_INITIATE_WAKE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_INITIATE_WAKE** structure indicates whether a processor requires an interrupt to wake up from an idle state.

Syntax

```
typedef struct _PEP_PPM_INITIATE_WAKE {
  BOOLEAN NeedInterruptForCompletion;
} PEP_PPM_INITIATE_WAKE, *PPEP_PPM_INITIATE_WAKE;
```

Members

`NeedInterruptForCompletion`

[out] Whether the processor requires an interrupt finish waking up from an idle state. Set to **TRUE** if the processor requires an interrupt, or to **FALSE** if it does not.

Remarks

This structure is used by the PEP_NOTIFY_PPM_INITIATE_WAKE notification. The **NeedInterruptForCompletion** member contains an output value that the platform extension plug-in (PEP) writes to the structure in response to this notification.

If the PEP sets the **NeedInterruptForCompletion** member to TRUE, the PEP must ensure that the processor is enabled to be interrupted before the AcceptProcessorNotification callback routine returns.

The PEP should set the **NeedInterruptForCompletion** member to **FALSE** if the processor is already running and/or will eventually exit the idle state (and is in the process of doing so) without requiring a software-generated interrupt.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

AcceptProcessorNotification

PEP_NOTIFY_PPM_INITIATE_WAKE

PEP_PPM_IS_PROCESSOR_HALTED structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IS_PROCESSOR_HALTED** structure indicates whether the processor is currently halted in its selected idle state.

Syntax

```
typedef struct _PEP_PPM_IS_PROCESSOR_HALTED {
  BOOLEAN Halted;
} PEP_PPM_IS_PROCESSOR_HALTED, *PPEP_PPM_IS_PROCESSOR_HALTED;
```

Members

`Halted`

[out] Whether the processor is halted. Set to **TRUE** if the processor is halted, or to **FALSE** if it is not halted.

Remarks

This structure is used by the PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED notification. The **Halted** member contains an output value that the platform extension plug-in (PEP) writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED

PEP_PPM_LPI_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.

Syntax

```
typedef struct _PEP_PPM_LPI_COMPLETE {
  ULONG  CoordinatedStateCount;
  PULONG CoordinatedStates;
} PEP_PPM_LPI_COMPLETE, *PPEP_PPM_LPI_COMPLETE;
```

Members

`CoordinatedStateCount`

`CoordinatedStates`

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification. The **Count** member of the structure contains an input value that PoFx supplies when this notification is sent. The **Counters** member contains an output value that the PEP writes in response to the notification. The PEP writes any additional **Counters** array elements to the output buffer area that follows the **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure. The buffer that PoFx allocated for this structure is guaranteed to be large enough to contain any array elements that follow the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h, Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

PEP_PROCESSOR_FEEDBACK_COUNTER

PEP_PPM_PARK_MASK structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_MASK** structure contains the current core parking mask.

Syntax

```
typedef struct _PEP_PPM_PARK_MASK {
  ULONG                     Count;
  ULONGLONG                 EvaluationTime;
  PPEP_PROCESSOR_PARK_STATE Processors;
} PEP_PPM_PARK_MASK, *PPEP_PPM_PARK_MASK;
```

Members

`Count`

[in] Indicates the number of processors in the **Processors** array.

`EvaluationTime`

[in] The interrupt time of the performance check evaluation that initiated this notification.

`Processors`

[in/out] An array of processors in the core parking domain.

On input

Indicates the OS parking preference for each processor.

On output

Returns the PEP parking preference for each processor.

The Processors array is guaranteed to contain the processors in increasing order by processor index.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_MASK notification

PEP_PROCESSOR_PARK_STATE

PEP_PPM_PARK_SELECTION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PARK_SELECTION** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PPM_PARK_SELECTION {
  ULONG                          AdditionalUnparkedProcessors;
  ULONG                          Count;
  PPEP_PROCESSOR_PARK_PREFERENCE Processors;
} PEP_PPM_PARK_SELECTION, *PPEP_PPM_PARK_SELECTION;
```

Members

`AdditionalUnparkedProcessors`

[in] The number of additional processors that need to be parked.

This number includes all processors in the **Processors** array for which the PEP sets **PepPreference** to PROCESSOR_PARK_PREFERENCE_PARKED but the operating system sets **PoPreference** to either PROCESSOR_PARK_PREFERENCE_UNPARKED or PROCESSOR_PARK_PREFERENCE_NONE. For more information about these values, see PEP_PROCESSOR_PARK_PREFERENCE.

This number does not include processors in the **Processors** array for which the operating system sets the **PoPreference** value to PROCESSOR_PARK_PREFERENCE_PARKED.

`Count`

[in] The number of elements in the array pointed to by the **Processors** member.

`Processors`

[in/out] A pointer to an array of PEP_PROCESSOR_PARK_PREFERENCE structures. The **Count** member specifies the number of elements in the array.

On input

Each array element indicates the parking preferences for a particular processor.

On output

Returns the PEP parking preference for each processor.

The Processors array is guaranteed to contain the processors in increasing order by processor index.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PARK_SELECTION notification. All three members of the structure contain input values that are set by the Windows power management framework (PoFx) before this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_SELECTION

PEP_PROCESSOR_PARK_PREFERENCE

PEP_PPM_PARK_SELECTION_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PARK_SELECTION_V2** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PPM_PARK_SELECTION_V2 {
  ULONG                          AdditionalUnparkedProcessors;
  ULONG                          Count;
  PPEP_PROCESSOR_PARK_PREFERENCE Processors;
  ULONGLONG                      EvaluationTime;
  UCHAR                          EvaluationType;
} PEP_PPM_PARK_SELECTION_V2, *PPEP_PPM_PARK_SELECTION_V2;
```

Members

`AdditionalUnparkedProcessors`

[in] The number of additional processors that need to be unparked.

This number includes all processors in the **Processors** array for which the PEP sets **PepPreference** to PROCESSOR_PARK_PREFERENCE_PARKED but the operating system sets **PoPreference** to either PROCESSOR_PARK_PREFERENCE_UNPARKED or PROCESSOR_PARK_PREFERENCE_NONE. For more information about these values, see PEP_PROCESSOR_PARK_PREFERENCE.

This number does not include processors in the **Processors** array for which the operating system sets the **PoPreference** value to PROCESSOR_PARK_PREFERENCE_PARKED.

`Count`

[in] The number of elements in the array pointed to by the **Processors** member.

`Processors`

[in/out] A pointer to an array of PEP_PROCESSOR_PARK_PREFERENCE structures. Each array element indicates the parking preferences for a particular processor. The **Count** member specifies the number of elements in the array.

`EvaluationTime`

The interrupt time of the performance check evaluation that initiated this notification.

`EvaluationType`

The type of evaluation being performed. This can be one of the following values:

VALUE	MEANING

PR OCE SSO R_P ARK _TY PE_ CO RE_ PAR KIN G 0x0 0	The processor should be parked.
PR OCE SSO R_P ARK _TY PE_I NTE RRU PT_ STE ERI NG 0x0 1	Interrupts should be steered away from the processor.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PARK_SELECTION_V2 notification notification. All three members of the structure contain input values that are set by the Windows power management framework (PoFx) before this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

PEP_PPM_PERF_CHECK_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_CHECK_COMPLETE** structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.

Syntax

```
typedef struct _PEP_PPM_PERF_CHECK_COMPLETE {
  ULONGLONG EvaluationTime;
} PEP_PPM_PERF_CHECK_COMPLETE, *PPEP_PPM_PERF_CHECK_COMPLETE;
```

Members

`EvaluationTime`

[in] The interrupt time of the performance check evaluation that initiated this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_CHECK_COMPLETE notification

PEP_PPM_PERF_SET structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_SET** structure specifies the new performance level that the operating system is requesting for the processor.

Syntax

```
typedef struct _PEP_PPM_PERF_SET {
  ULONG MinimumPerformance;
  ULONG MaximumPerformance;
  ULONG DesiredPerformance;
  ULONG TimeWindow;
  ULONG PerformanceTolerance;
} PEP_PPM_PERF_SET, *PPEP_PPM_PERF_SET;
```

Members

`MinimumPerformance`

[in] The new minimum performance level in platform-specific units. This member indicates the absolute minimum performance level that the processor can run at.

`MaximumPerformance`

[in] The new maximum performance level in platform-specific units. This member indicates the absolute maximum performance level that the processor can run at.

`DesiredPerformance`

[in] The new desired performance level in platform-specific units. This member is set to a value in the range **MinimumPerformance** to **MaximumPerformance**.

`TimeWindow`

[in] The width, in milliseconds, of the new time window over which the platform must provide the required average performance. If this time window is nonzero, the instantaneous performance provided by the platform does not need to match the desired performance level, and only needs to fall within the constraints specified by the minimum and maximum performance level.

`PerformanceTolerance`

[in] The new performance tolerance in platform-specific units. This member contains the minimum performance level that the platform can deliver and still meet the operating system's performance requirements.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PERF_SET notification. All five members of this structure contain input values that PoFx supplies when this notification is sent to the PEP.

Processor performance levels are specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done. For more information, see Platform Performance Thresholds.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_SET

Platform Performance Thresholds

PEP_PPM_PERF_SET_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_PERF_SET** notification at runtime to set the current operating performance of the processor.

Syntax

```
typedef struct _PEP_PPM_PERF_SET_STATE {
  UCHAR MinimumPerformanceState;
  UCHAR MaximumPerformanceState;
  UCHAR DesiredPerformanceState;
} PEP_PPM_PERF_SET_STATE, *PPEP_PPM_PERF_SET_STATE;
```

Members

`MinimumPerformanceState`

On input, the new requested minimum performance state. This state is defined as the absolute minimum instantaneous performance level at which the processor may run. The operating system may set this value to any performance value in the range [Lowest Performance, Guaranteed Performance], inclusive. Minimum performance must never be set to a value higher than maximum performance.

`MaximumPerformanceState`

On input, the new requested maximum performance state. This state is defined as the absolute maximum instantaneous performance level at which the processor may run. The OS may set Maximum performance to any performance value in the range [Lowest Performance, Highest Performance], inclusive.

`DesiredPerformanceState`

On input, the new requested desired performance state. This state is defined as the performance level power manager is requesting from the processor. Desired performance may be set to any performance value in the range [Minimum Performance, Maximum Performance], inclusive. Desired performance may take one of two meanings, depending on whether the desired performance is above or below the guaranteed performance level.

- Below the guaranteed performance level, desired performance expresses the average performance level the platform must provide.
- Above the guaranteed performance level, the platform must provide the guaranteed performance level. The platform should attempt to provide up to the desired performance level, if current operating conditions allow for it, but it is not required to do so.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Header	pep_x.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_PLATFORM_STATE_RESIDENCIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PLATFORM_STATE_RESIDENCIES** structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.

Syntax

```
typedef struct _PEP_PPM_PLATFORM_STATE_RESIDENCIES {
  ULONG                              Count;
  PPEP_PPM_PLATFORM_STATE_RESIDENCY States;
} PEP_PPM_PLATFORM_STATE_RESIDENCIES, *PPEP_PPM_PLATFORM_STATE_RESIDENCIES;
```

Members

`Count`

[in] The number of elements in the array pointed to by the **States** member. The number of array elements equals the number of platform idle states that the PEP previously provided in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

`States`

[in] A pointer to an output buffer to which the PEP writes an array of PEP_PPM_PLATFORM_STATE_RESIDENCY structures. Each structure contains the accumulated residency and transition count information for a particular platform idle state. This buffer is guaranteed to be large enough to contain the number of array elements specified by the **Count** member. Each idle state is identified by its array index. If the array contains N elements, idle state indexes range from 0 to N-1.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES notification. Both members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent to the PEP. PoFx allocates the output buffer pointed to by the **States** member, and the PEP writes to this buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES

PEP_PPM_PLATFORM_STATE_RESIDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PLATFORM_STATE_RESIDENCY** structure specifies the accumulated residency time and transition count for a particular platform idle state.

Syntax

```
typedef struct _PEP_PPM_PLATFORM_STATE_RESIDENCY {
  ULONG64 Residency;
  ULONG64 TransitionCount;
} PEP_PPM_PLATFORM_STATE_RESIDENCY, *PPEP_PPM_PLATFORM_STATE_RESIDENCY;
```

Members

`Residency`

The accumulated residency time, in 100-nanosecond units. This is the total amount of time that the hardware platform has spent in this idle state since system startup.

`TransitionCount`

The total number of times that the platform has entered this idle state since system startup.

Remarks

The **States** member of the PEP_PPM_PLATFORM_STATE_RESIDENCIES structure is a pointer to an array of **PEP_PPM_PLATFORM_STATE_RESIDENCY** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_PPM_PLATFORM_STATE_RESIDENCIES

PEP_PPM_QUERY_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_CAPABILITIES** structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_PPM_QUERY_CAPABILITIES {
  ULONG   FeedbackCounterCount;
  ULONG   IdleStateCount;
  BOOLEAN PerformanceStatesSupported;
  BOOLEAN ParkingSupported;
  UCHAR   DiscretePerformanceStateCount;
} PEP_PPM_QUERY_CAPABILITIES, *PPEP_PPM_QUERY_CAPABILITIES;
```

Members

`FeedbackCounterCount`

[out] The number of processor performance feedback counters supported by the PEP for this processor. On x86/AMD64 platforms, this must be zero.

`IdleStateCount`

[out] The number of processor idle states that the PEP supports. The PEP is not required to support the same number of idle states for all processors. The PEP can set **IdleStateCount** = 0 to indicate that it doesn't support any idle states, in which case the PEP doesn't need to implement any other notifications that deal with processor idle states.

`PerformanceStatesSupported`

[out] Whether the PEP supports performance state controls. Set to TRUE if the PEP supports performance states, or to FALSE if it does not.

`ParkingSupported`

[out] Whether the PEP supports supplying core parking hints. Set to TRUE if the PEP can indicate which cores to park, or to FALSE if it cannot.

`DiscretePerformanceStateCount`

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification. All four members contain output values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_COORDINATED_DEPENDENCY** structure describes dependencies for coordinated idle states.

Syntax

```
typedef struct _PEP_PPM_QUERY_COORDINATED_DEPENDENCY {
  ULONG                              StateIndex;
  ULONG                              DependencyIndex;
  ULONG                              DependencySize;
  ULONG                              DependencySizeUsed;
  POHANDLE                           TargetProcessor;
  PEP_COORDINATED_DEPENDENCY_OPTION Options[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_COORDINATED_DEPENDENCY, *PPEP_PPM_QUERY_COORDINATED_DEPENDENCY;
```

Members

`StateIndex`

[in] The index of the coordinated idle state which is having its dependencies queried.

`DependencyIndex`

[in] The index of the dependency being queried.

`DependencySize`

[in] The size of the **Dependencies** array.

`DependencySizeUsed`

[out] The number of elements of the **Dependencies** array filled in by the PEP.

`TargetProcessor`

[out] The **POHANDLE** corresponding to the processor being targeted by this dependency, or **NULL** if this is a coordinated state dependency.

`Options`

[out] A list of PEP_COORDINATED_DEPENDENCY_OPTION structures describing dependency options, one of which must be satisfied for this coordinated state to be entered.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

See Also

PEP_COORDINATED_DEPENDENCY_OPTION

PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY notification

PEP_PPM_QUERY_COORDINATED_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_COORDINATED_STATES** structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_COORDINATED_STATES {
  ULONG                      Count;
  PEP_COORDINATED_IDLE_STATE States[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_COORDINATED_STATES, *PPEP_PPM_QUERY_COORDINATED_STATES;
```

Members

`Count`

[in] The size of the **States** array. This will be equal to the **PlatformStateCount** returned from the query platform states notification.

`States`

[out] An array of PEP_COORDINATED_IDLE_STATE structures that contains information about each coordinated idle state that the PEP supports.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COORDINATED_IDLE_STATE

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification

PEP_PPM_QUERY_DISCRETE_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES** notification that stores the list of discrete performance states that PEP supports, if the **PEP_NOTIFY_PPM_QUERY_CAPABILITIES** notification indicates support for discrete performance states.

Syntax

```
typedef struct _PEP_PPM_QUERY_DISCRETE_PERF_STATES {
  ULONG                     Count;
  PPEP_PROCESSOR_PERF_STATE States;
} PEP_PPM_QUERY_DISCRETE_PERF_STATES, *PPEP_PPM_QUERY_DISCRETE_PERF_STATES;
```

Members

`Count`

On input, the size of the array pointed to by *States*.

`States`

On output, an array of PEP_PROCESSOR_PERF_STATE structures that indicates performance states that is filled by PEP.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pep_x.h (include Pep_x.h)

See Also

PEP_PPM_QUERY_CAPABILITIES

Processor power management (PPM) notifications

PEP_PPM_QUERY_DOMAIN_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO** notification that queries for information about a performance domain.

Syntax

```
typedef struct _PEP_PPM_QUERY_DOMAIN_INFO {
  ULONG   DomainId;
  UCHAR   CoordinationType;
  BOOLEAN IdleProcessorsDiscounted;
  BOOLEAN SchedulerDirectedTransitionsSupported;
  BOOLEAN AffinitizePerfSet;
  ULONG   WorstCaseTransitionLatency;
  ULONG   WorstCaseTransitionOverhead;
} PEP_PPM_QUERY_DOMAIN_INFO, *PPEP_PPM_QUERY_DOMAIN_INFO;
```

Members

`DomainId`

On input, the domain ID of the domain being queried.

`CoordinationType`

On output, indicates the coordination type of the performance domain. The possible values are:

- PROCESSOR_DOMAIN_COORDIANTION_SW_ALL (0x00)
- PROCESSOR_DOMAIN_COORDIANTION_SW_ANY (0x01)
- PROCESSOR_DOMAIN_COORDIANTION_HW_ALL (0x02)

If this notification is not supported, PROCESSOR_DOMAIN_COORDIANTION_SW_ALL coordination is used.

`IdleProcessorsDiscounted`

On output, and if HW-ALL coordination is used, this value is TRUE if the platform automatically ignores requests from idle processors as it determines the resolved performance level of the domain, FALSE if requests from idle processors continue to be honored.
If PROCESSOR_DOMAIN_COORDIANTION_HW_ALL coordination is not used: this field is ignored.

`SchedulerDirectedTransitionsSupported`

On output, this value is TRUE if the PEP supports the requirements of scheduler directed performance transitions for this domain, FALSE otherwise.

`AffinitizePerfSet`

`WorstCaseTransitionLatency`

On output, this value indicates the worst case latency for the processor to transition to a new performance state, measured from the time the OS issues the notification to the PEP to transition to a new performance level, to the time the processor is running at the new performance level, in 100ns units.

`WorstCaseTransitionOverhead`

On output, this value indicates the worst case overhead (PEP code execution time plus time the processor is not executing instructions) measured from the time the OS issues the notification to the PEP to transition to a new performance level, to the time the processor is running at the new performance level, in 100ns units.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pep_x.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_QUERY_FEEDBACK_COUNTERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.

Syntax

```
typedef struct _PEP_PPM_QUERY_FEEDBACK_COUNTERS {
  ULONG                          Count;
  PEP_PROCESSOR_FEEDBACK_COUNTER Counters[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_FEEDBACK_COUNTERS, *PPEP_PPM_QUERY_FEEDBACK_COUNTERS;
```

Members

`Count`

[in] The number of processor performance feedback counters supported by the PEP. The PEP previously supplied this count in response to a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification.

`Counters`

[out] The first element in an array of PEP_PROCESSOR_FEEDBACK_COUNTER structures. If this array contains more than one element, the additional elements immediately follow the end of the **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure. The **Count** member specifies the number of elements in this array. For more information, see Remarks.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification. The **Count** member of the structure contains an input value that PoFx supplies when this notification is sent. The **Counters** member contains an output value that the PEP writes in response to the notification. The PEP writes any additional **Counters** array elements to the output buffer area that follows the **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure. The buffer that PoFx allocated for this structure is guaranteed to be large enough to contain any array elements that follow the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h, Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

PEP_PROCESSOR_FEEDBACK_COUNTER

PEP_PPM_QUERY_IDLE_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_IDLE_STATES** structure describes the idle states of a particular processor.

Syntax

```
typedef struct _PEP_PPM_QUERY_IDLE_STATES {
  ULONG                    Count;
  ULONG                    MaximumCoordinatedProcessors;
  PEP_PROCESSOR_IDLE_STATE IdleStates[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_IDLE_STATES, *PPEP_PPM_QUERY_IDLE_STATES;
```

Members

`Count`

[in] The number of elements in the **IdleStates** array. This member is set to the **IdleStateCount** value that the PEP previously supplied for this processor in response to the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification.

`MaximumCoordinatedProcessors`

[out] The maximum number of secondary processors that are subordinate to this processor for any transitions to idle states. A primary processor can enter the selected idle state only after the secondary processors have entered their corresponding idle states. The PEP must set the **MaximumCoordinatedProcessors** member to a number that is less than the total number of processors in the hardware platform.

`IdleStates`

[in] The first element in an array of PEP_PROCESSOR_IDLE_STATE structures. Each array element describes one of the idle states of this processor. If the array contains more than one element, the additional elements immediately follow the end of the **PEP_PPM_QUERY_IDLE_STATES** structure. The number of array elements is specified by the **Count** member. The buffer that PoFx allocates to hold this structure is guaranteed to be large enough to contain the **PEP_PPM_QUERY_IDLE_STATES** structure plus any array elements that follow this structure. Each idle state is identified by its array index. If the array contains N elements, idle state indexes range from 0 to N−1.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification. The **Count** member of the structure contains an input value that PoFx supplies when this notification is sent. The other two members contain output values that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_NOTIFY_PPM_QUERY_IDLE_STATES

PEP_PROCESSOR_IDLE_STATE

PEP_PPM_QUERY_IDLE_STATES_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_IDLE_STATES_V2** structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_IDLE_STATES_V2 {
  ULONG                       Count;
  PEP_PROCESSOR_IDLE_STATE_V2 IdleStates[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_IDLE_STATES_V2, *PPEP_PPM_QUERY_IDLE_STATES_V2;
```

Members

`Count`

[in] The number of elements in the **IdleStates** array. The Windows power management framework (PoFx) previously sent the PEP a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification to determine this number.

`IdleStates`

[out] The first element in an array of PEP_PROCESSOR_IDLE_STATE_V2 structures that describe the processor idle states. If this array contains more than one element, the addition array elements follow the end of the **PEP_PPM_QUERY_IDLE_STATES_V2** structure.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. The **Count** member of the **PEP_PPM_QUERY_IDLE_STATES_V2** structure contains an input value that is supplied by the Windows power management framework (PoFx). The **IdleStates** array elements contain output values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_IDLE_STATE_V2

PEP_PPM_QUERY_LP_SETTINGS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_LP_SETTINGS** structure contains a kernel handle to the registry key that contains the power optimization settings that the platform extension plug-in (PEP) has defined for each power scenario.

Syntax

```
typedef struct _PEP_PPM_QUERY_LP_SETTINGS {
  HANDLE RegistryRoot;
} PEP_PPM_QUERY_LP_SETTINGS, *PPEP_PPM_QUERY_LP_SETTINGS;
```

Members

`RegistryRoot`

[out] The kernel handle to the registry root that contains the settings for a platform power scenario. For more information, see Remarks.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_LP_SETTINGS notification. The **RegistryRoot** member contains an output value that the PEP writes to the structure in response to this notification.

The PEP can provide settings for certain key power scenarios such as full-screen video playback and connected standby that have unique characteristics that can benefit from specially tuned power policies that are different from the default settings used by Windows. The PEP loads the settings for such a power scenario into the registry and passes the registry key for these settings to the operating system.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_NOTIFY_PPM_QUERY_LP_SETTINGS

PEP_PPM_QUERY_PERF_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PERF_CAPABILITIES** structure describes the performance capabilities of the processors in the specified processor performance domain.

Syntax

```
typedef struct _PEP_PPM_QUERY_PERF_CAPABILITIES {
  ULONG HighestPerformance;
  ULONG NominalPerformance;
  ULONG LowestNonlinearPerformance;
  ULONG LowestPerformance;
  ULONG DomainId;
  ULONG DomainMembers;
} PEP_PPM_QUERY_PERF_CAPABILITIES, *PPEP_PPM_QUERY_PERF_CAPABILITIES;
```

Members

`HighestPerformance`

[out] The highest performance level in platform-specific units. For more information, see Remarks.

`NominalPerformance`

[out] The nominal performance level in platform-specific units. For more information, see Remarks.

`LowestNonlinearPerformance`

[out] The lowest nonlinear performance level in platform-specific units. For more information, see Remarks.

`LowestPerformance`

[out] The lowest performance level in platform-specific units. For more information, see Remarks.

`DomainId`

[out] The domain ID of the processor performance domain.

`DomainMembers`

[out] The number of processors in this performance domain.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification. All six members contain output values that the platform extension plug-in (PEP) writes to the structure in response to this notification.

Processor performance levels are specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done. For more information, see Platform Performance Thresholds.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES

Platform Performance Thresholds

PEP_PPM_PERF_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_CONSTRAINTS** structure describes the performance limits to apply to the processor.

Syntax

```
typedef struct _PEP_PPM_QUERY_PERF_CONSTRAINTS {
  ULONG GuaranteedPerformanceLimit;
  ULONG LimitReasons;
} PEP_PPM_PERF_CONSTRAINTS, *PPEP_PPM_PERF_CONSTRAINTS;
```

Members

`GuaranteedPerformanceLimit`

[out] The guaranteed performance threshold in platform-specific units. If nonzero, this value specifies the maximum sustained performance level of the processor, taking into account all known external constraints (power budgeting, thermal constraints, power source, and so on). All processors are expected to be able to simultaneously sustain their guaranteed performance levels. For more information, see Platform Performance Thresholds.

`LimitReasons`

[out] The reasons for which the processor's performance can be limited. Set this member to zero or to the bitwise-OR of one or more of the following values:

LIMIT REASON	MEANING
PERFORMANCE_LIMIT_THERMAL 0x1	The processor performance can be limited due to an out-of-band (that is, non-ACPI Thermal Zone) thermal condition.
PERFORMANCE_LIMIT_POWER 0x2	The processor performance can be limited to keep power consumption of the processor within hardware safety limits.

PER FOR MA NCE _LI MIT _DO MAI N_D EPE NDE NC Y 0x4	The processor performance can be limited because of a voltage or frequency dependency between this processor and another, non-processor device.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PERF_CONSTRAINTS notification. The **GuaranteedPerformanceLimit** and **LimitReasons** members contain output values that the platform extension plug-in (PEP) writes to the structure in response to this notification.

The **GuaranteedPerformanceLimit** member value is specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_CONSTRAINTS

PEP_PPM_QUERY_PLATFORM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PLATFORM_STATE** structure contains information about a platform idle state.

Syntax

```
typedef struct _PEP_PPM_QUERY_PLATFORM_STATE {
  ULONG                  StateIndex;
  PEP_PLATFORM_IDLE_STATE State;
} PEP_PPM_QUERY_PLATFORM_STATE, *PPEP_PPM_QUERY_PLATFORM_STATE;
```

Members

`StateIndex`

[in] The index of this platform idle state. If the hardware platform supports N platform idle states, the states are numbered 0 to N-1. The Windows power management framework (PoFx) previously sent a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification to the platform extension plug-in (PEP) to determine the number of supported platform idle states.

`State`

[out] A PEP_PLATFORM_IDLE_STATE structure that describes the platform idle state.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification. The **StateIndex** member of the **PEP_PPM_QUERY_PLATFORM_STATE** structure contains an input value that is supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **State** member contains an output value that the PEP writes to the structure in response to the notification.

The buffer that PoFx allocates to hold the **PEP_PPM_QUERY_PLATFORM_STATE** structure is guaranteed to be large enough to contain this structure plus any elements of the **State.DependencyArray** array that follow this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PLATFORM_IDLE_STATE

PEP_PPM_QUERY_PLATFORM_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PLATFORM_STATES** structure specifies the number of platform idle states the hardware platform supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_PLATFORM_STATES {
  ULONG PlatformStateCount;
} PEP_PPM_QUERY_PLATFORM_STATES, *PPEP_PPM_QUERY_PLATFORM_STATES;
```

Members

`PlatformStateCount`

The number of platform idle states supported by the platform.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_QUERY_STATE_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_STATE_NAME** structure contains information about a specific coordinated or platform idle state.

Syntax

```
typedef struct _PEP_PPM_QUERY_STATE_NAME {
  ULONG  StateIndex;
  USHORT NameSize;
  PWSTR  Name;
} PEP_PPM_QUERY_STATE_NAME, *PPEP_PPM_QUERY_STATE_NAME;
```

Members

`StateIndex`

[in] The index of the state that the OS is querying about.

`NameSize`

[in/out] If **Name** is **NULL**, returns the size of the buffer required to hold **Name** (including the trailing **UNICODE_NULL** character), in WCHARs. If **Name** is not **NULL**, supplies the size of the **Name** buffer, in WCHARs.

`Name`

[in] A pointer to a buffer to copy the name of the state to, or **NULL**.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATE_NAME notification

PEP_NOTIFY_PPM_QUERY_PROCESSOR_STATE_NAME notification

PEP_PPM_QUERY_VETO_REASON structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_VETO_REASON** structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.

Syntax

```
typedef struct _PEP_PPM_QUERY_VETO_REASON {
  ULONG  VetoReason;
  USHORT NameSize;
  PWSTR  Name;
} PEP_PPM_QUERY_VETO_REASON, *PPEP_PPM_QUERY_VETO_REASON;
```

Members

`VetoReason`

[in] The index that identifies this veto reason. If the platform extension plug-in (PEP) supports N veto reasons, valid veto reason indexes range from 1 to N. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification.

`NameSize`

[out] The size, in bytes, of the string buffer pointed to by the **Name** member. This string size includes the storage space for the null character that terminates the wide-character string.

`Name`

[in] A pointer to a PoFx-allocated string buffer. If this buffer pointer is non-**NULL**, the PEP writes a wide-character, null-terminated string to the buffer (and writes nothing to the **NameSize** member). Otherwise, the PEP writes nothing to this buffer and instead writes the required string size to the **NameSize** member.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_VETO_REASON notification. The **VetoReason** and **Name** members contain input values that are supplied by the Windows power management framework (PoFx). If the input value of the **Name** member is **NULL**, the PEP writes an output value (the required string buffer size) to the **NameSize** member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_VETO_REASON

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_PPM_QUERY_VETO_REASONS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_VETO_REASONS** structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.

Syntax

```
typedef struct _PEP_PPM_QUERY_VETO_REASONS {
  ULONG VetoReasonCount;
} PEP_PPM_QUERY_VETO_REASONS, *PPEP_PPM_QUERY_VETO_REASONS;
```

Members

`VetoReasonCount`

[out] The number of veto codes used by the PEP.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. The **VetoReasonCount** member contains an output value that the PEP writes to this member in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_PPM_RESUME_FROM_SYSTEM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used by the **PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE** notification that notifies the PEP that the system has just resumed from a system power state.

Syntax

```
typedef struct _PEP_PPM_RESUME_FROM_SYSTEM_STATE {
  SYSTEM_POWER_STATE TargetState;
} PEP_PPM_RESUME_FROM_SYSTEM_STATE, *PPEP_PPM_RESUME_FROM_SYSTEM_STATE;
```

Members

`TargetState`

On input, a SYSTEM_POWER_STATE-type value that indicates the power state that the system is entering.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pep_x.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_TEST_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_TEST_IDLE_STATE** structure contains information about whether the processor can immediately enter a processor idle state.

Syntax

```
typedef struct _PEP_PPM_TEST_IDLE_STATE {
  ULONG ProcessorState;
  ULONG PlatformState;
  ULONG VetoReason;
} PEP_PPM_TEST_IDLE_STATE, *PPEP_PPM_TEST_IDLE_STATE;
```

Members

`ProcessorState`

(input) The index of the processor idle state that is to be entered. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`VetoReason`

[out] The nonzero index value that identifies reason the PEP needs to veto this idle state transition, or **PEP_IDLE_VETO_NONE** (0) if the PEP is prepared to immediately make this transition. If the PEP supports N veto reasons, valid veto reason indexes range from 1 to N. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification.

Remarks

This structure is used by the PEP_NOTIFY_PPM_TEST_IDLE_STATE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The PEP writes an output value to the **VetoReason** member of this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_NOTIFY_PPM_TEST_IDLE_STATE

PEP_PREPARE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PREPARE_DEVICE** structure identifies a device that must be started up in preparation for its use by the operating system.

Syntax

```
typedef struct _PEP_PREPARE_DEVICE {
  PCUNICODE_STRING DeviceId;
  BOOLEAN          DeviceAccepted;
} PEP_PREPARE_DEVICE, *PPEP_PREPARE_DEVICE;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device. The PEP sets this member to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device. The PEP that claims ownership is responsible for handling device power management (DPM) notifications for the device.

Remarks

This structure is used by the PEP_DPM_PREPARE_DEVICE notification. The **DeviceId** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_PREPARE_DEVICE

UNICODE_STRING

PEP_PROCESSOR_FEEDBACK_COUNTER structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_FEEDBACK_COUNTER** structure describes a feedback counter to the operating system.

Syntax

```
typedef struct _PEP_PROCESSOR_FEEDBACK_COUNTER {
  struct {
    ULONG Affinitized : 1;
    ULONG Type : 2;
    ULONG Counter : 4;
    ULONG DiscountIdle : 1;
    ULONG Reserved : 24;
  };
  ULONG  NominalRate;
} PEP_PROCESSOR_FEEDBACK_COUNTER, *PPEP_PROCESSOR_FEEDBACK_COUNTER;
```

Members

`Affinitized`

Identifies the counter process affinity. If set to 1, the counter must be read while executing on the target processor, otherwise, it will be set to 0.

`Type`

Specifies the data type of the counter.

The processor feedback counter data types are:

VALUE	MEANING
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_T** **YPE** **_IN** **STA** **NT** **ANE** **OU** **S** 0x0 0	The feedback counter returns the instantaneous value of the property being counted.

PR **OCE** **SSO** **R_F** **EED** **BAC** **K_T** **YPE** **_RE** **LAT** **IVE** 0x0 1	The feedback counter returns two incrementing values: the nominal count, and the actual count. Nominal count increments at a fixed nominal rate. Actual count increments at a variable rate relative to the property being counted. When the property is incrementing at its nominal rate, the two values should increment at the same rate. To compute an average rate over a time period, the OS reads the counter once and the beginning of the period and once at the end and computes:

`Counter`

Specifies the data the counter is providing.

The processor feedback counter types are:

VALUE	MEANING
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_C** **OU** **NTE** **R_F** **REQ** **UEN** **CY** 0x0 0	The feedback counter returns the clock speed of the processor. The nominal rate is the nominal clock speed, in MHz.
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_C** **OU** **NTE** **R_P** **ERF** **OR** **MA** **NCE** 0x0 1	The feedback counter returns the current processor performance. The nominal rate is equivalent to the processor's **NominalPerformance** (see PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification).

`DiscountIdle`

`Reserved`

This member is reserved and should be set to zero.

`NominalRate`

Specifies the nominal rate of the counter.

Remarks

This structure

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP structures

PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification

PEP_PROCESSOR_IDLE_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_CONSTRAINTS** structure specifies a set of constraints that the PEP uses to select a processor idle state.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_CONSTRAINTS {
  ULONGLONG               IdleDuration;
  BOOLEAN                 Interruptible;
  PEP_PROCESSOR_IDLE_TYPE Type;
} PEP_PROCESSOR_IDLE_CONSTRAINTS, *PPEP_PROCESSOR_IDLE_CONSTRAINTS;
```

Members

`IdleDuration`

The length of time, in 100-nanosecond units, for which the processor is expected to stay in the selected idle state. This is the operating system's current best estimate and might vary over time without further notification.

`Interruptible`

Whether the processor must be able to respond to interrupts in the selected idle state. A value of TRUE indicates that the processor must be interruptible in this idle state. A value of FALSE indicates that the processor is not required to be interruptible in this idle state.

`Type`

A PEP_PROCESSOR_IDLE_TYPE enumeration value that indicates whether these idle constraints apply to just the current processor (**Type** = **PepIdleTypeProcessor**) or to all processors in the hardware platform (**Type** = **PepIdleTypePlatform**).

Remarks

The **Constraints** member of the PEP_PPM_IDLE_SELECT structure is a pointer to a **PEP_PROCESSOR_IDLE_CONSTRAINTS** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_PPM_IDLE_SELECT

PEP_PROCESSOR_IDLE_TYPE

PEP_PROCESSOR_IDLE_DEPENDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_DEPENDENCY** structure specifies the dependencies of a platform idle state on the specified processor.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_DEPENDENCY {
  POHANDLE TargetProcessor;
  UCHAR    ExpectedState;
  BOOLEAN  AllowDeeperStates;
  BOOLEAN  LooseDependency;
} PEP_PROCESSOR_IDLE_DEPENDENCY, *PPEP_PROCESSOR_IDLE_DEPENDENCY;
```

Members

`TargetProcessor`

A **POHANDLE** value that identifies the processor. The platform extension plug-in (PEP) received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`ExpectedState`

The index of the processor idle state that this processor must enter to initiate the platform's transition to this platform idle state. If the **IdleStates** array in the PEP_PPM_QUERY_IDLE_STATES_V2 structure contains N elements, the idle states are numbered 0 to N-1 in the order in which they appear in the array.

`AllowDeeperStates`

Whether a secondary (subordinate) processor can be in a deeper processor idle state than **ExpectedState** and still satisfy the constraints of the transition to the platform idle state. Set this member to **TRUE** if a secondary processor in a deeper idle state can still satisfy the constraints, and to **FALSE** if it cannot. A primary processor (identified by the **TargetProcessor** member) can enter the selected idle state only after its secondary processors have entered their corresponding idle states.

`LooseDependency`

Whether the platform idle state has a loose dependency on the idle state of this processor. Set this member to **TRUE** to indicate a loose dependency in which a best effort to synchronize the idle state transitions of the platform and processor is sufficient. Set to **FALSE** to indicate a dependency in which the operating system is required to strictly synchronize these transitions. If **LooseDependency** is **FALSE**, the **WakesSpuriously** bit in the PEP_PROCESSOR_IDLE_STATE_V2 structure for this processor must be **FALSE**.

Remarks

The **DependencyArray** member of the PEP_PPM_IDLE_SELECT structure is a pointer to an array of **PEP_PROCESSOR_IDLE_DEPENDENCY** structures. The **DependencyArray** member of the PEP_PLATFORM_IDLE_STATE structure is the first element in an array of **PEP_PROCESSOR_IDLE_DEPENDENCY** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_PLATFORM_IDLE_STATE

PEP_PPM_IDLE_SELECT

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_IDLE_STATE_V2

PEP_PROCESSOR_IDLE_STATE structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_STATE** structure describes the capabilities of a processor idle state.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_STATE {
  union {
    ULONG Ulong;
    struct {
      ULONG Interruptible : 1;
      ULONG CacheCoherent : 1;
      ULONG ThreadContextRetained : 1;
      ULONG CStateType : 4;
      ULONG Reserved : 25;
    };
  };
} PEP_PROCESSOR_IDLE_STATE, *PPEP_PROCESSOR_IDLE_STATE;
```

Members

`Ulong`

`Interruptible`

`CacheCoherent`

`ThreadContextRetained`

`CStateType`

`Reserved`

Remarks

The **IdleStates** member of the PEP_PPM_QUERY_IDLE_STATES structure is the first element in an array of **PEP_PROCESSOR_IDLE_STATE** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_PPM_QUERY_IDLE_STATES

PEP_PROCESSOR_IDLE_STATE_UPDATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure contains the updated properties of a processor idle state.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_STATE_UPDATE {
  ULONG Version;
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PROCESSOR_IDLE_STATE_UPDATE, *PPEP_PROCESSOR_IDLE_STATE_UPDATE;
```

Members

`Version`

The version number of this structure. Set this member to PEP_PROCESSOR_IDLE_STATE_UPDATE_VERSION.

`Latency`

The worst-case latency, in 100-nanosecond units, that the platform requires to wake from this platform idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. PoFx uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

The *Update* parameter to the UpdateProcessorIdleState routine is a pointer to a **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure. This routine updates the properties of the specified idle state for a processor. The PEP must not call this routine until it has responded to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification for this processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

UpdateProcessorIdleState

PEP_PROCESSOR_IDLE_STATE_V2 structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_STATE_V2** structure describes a processor idle state that the platform extension plug-in (PEP) supports.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_STATE_V2 {
  union {
    ULONG Ulong;
    struct {
      ULONG Interruptible : 1;
      ULONG CacheCoherent : 1;
      ULONG ThreadContextRetained : 1;
      ULONG CStateType : 4;
      ULONG WakesSpuriously : 1;
      ULONG PlatformOnly : 1;
      ULONG Autonomous : 1;
      ULONG Reserved : 22;
    };
  };
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PROCESSOR_IDLE_STATE_V2, *PPEP_PROCESSOR_IDLE_STATE_V2;
```

Members

`Ulong`

`Interruptible`

`CacheCoherent`

`ThreadContextRetained`

`CStateType`

`WakesSpuriously`

`PlatformOnly`

`Autonomous`

`Reserved`

`Latency`

The worst-case latency, in 100-nanosecond units, that the processor requires to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the processor must spend in this idle state to make a transition to this state worthwhile. The Windows power management framework (PoFx) uses this member value as a hint to avoid switching a processor to an idle state unless the processor is likely to remain in

this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. The **IdleStates** member of the PEP_PPM_QUERY_IDLE_STATES_V2 structure is the first element in an array of **PEP_PROCESSOR_IDLE_STATE_V2** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_PARK_PREFERENCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_PREFERENCE** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PROCESSOR_PARK_PREFERENCE {
  PEPHANDLE Processor;
  UCHAR     PoPreference;
  UCHAR     PepPreference;
} PEP_PROCESSOR_PARK_PREFERENCE, *PPEP_PROCESSOR_PARK_PREFERENCE;
```

Members

`Processor`

A **PEPHANDLE** value that identifies the processor. The PEP supplied this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`PoPreference`

The operating system's parking preference for this processor. For more information, see Remarks.

`PepPreference`

The PEP's parking preference for this processor. For more information, see Remarks.

Remarks

The **Processors** member of the PEP_PPM_PARK_SELECTION structure is a pointer to an array of **PEP_PROCESSOR_PARK_PREFERENCE** structures.

The **Processor** and **PoPreference** members of the **PEP_PROCESSOR_PARK_PREFERENCE** structure contain input values that are supplied by the Windows power management framework when a PEP_NOTIFY_PPM_PARK_SELECTION notification is sent to the PEP. The **PepPreference** member contains an output value that the PEP writes to the structure in response to this notification.

The following table lists the three values that are defined for the **PoPreference** and **PepPreference** members.

PARKING PREFERENCE	VALUE	MEANING
PROCESSOR_PARK_PREFERENCE_NONE	0x0	No preference. Either parked or unparked is okay.
PROCESSOR_PARK_PREFERENCE_PARKED	0x1	The operating system or PEP specifies that the processor should be parked.

PROCESSOR_PARK_PREFERENCE_UNPARKED	0x2	The operating system or PEP specifies that the processor should not be parked.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_PARK_SELECTION

PEP_PPM_PARK_SELECTION

PEP_PROCESSOR_PARK_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_STATE** structure describes the parking state for a single processor.

Syntax

```
typedef struct _PEP_PROCESSOR_PARK_STATE {
  PEPHANDLE Processor;
  BOOLEAN   Parked;
  UCHAR     Reserved[3];
} PEP_PROCESSOR_PARK_STATE, *PPEP_PROCESSOR_PARK_STATE;
```

Members

`Processor`

Specifies the **PEPHANDLE** associated with this processor.

`Parked`

Specifies whether or not this processor is parked.

`Reserved`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_MASK notification

PEP_PPM_PARK_MASK

PEP_PROCESSOR_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Use in the **PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES** notification. This structure describes the properties of a single performance state.

Syntax

```
typedef struct _PEP_PROCESSOR_PERF_STATE {
  ULONG Performance;
  ULONG Frequency;
  ULONG Reserved[4];
} PEP_PROCESSOR_PERF_STATE, *PPEP_PROCESSOR_PERF_STATE;
```

Members

`Performance`

On output, the performance level (on the scale returned by the PEP_QUERY_COMPONENT_PERF_CAPABILITIES notification) of this performance state.

`Frequency`

On output, the frequency (in MHz) of this performance state.

`Reserved`

Reserved.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pep_x.h (include Pep_x.h)

See Also

PEP_PPM_QUERY_DISCRETE_PERF_STATES

Processor power management (PPM) notifications

PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_CAPABILITIES** structure specifies the number of performance state (P-state) sets that are defined for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_CAPABILITIES {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     SetCount;
} PEP_QUERY_COMPONENT_PERF_CAPABILITIES, *PPEP_QUERY_COMPONENT_PERF_CAPABILITIES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`SetCount`

[out] The number of P-state sets that are defined for this component.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification. The **DeviceHandle** and **Component** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent. The **SetCount** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_SET** structure contains query information about a set of performance state values (P-state set) for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_SET {
  PEPHANDLE           DeviceHandle;
  ULONG               Component;
  ULONG               Set;
  ULONGLONG           Flags;
  PEP_PERF_STATE_UNIT Unit;
  PEP_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG Count;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PEP_QUERY_COMPONENT_PERF_SET, *PPEP_QUERY_COMPONENT_PERF_SET;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If this component has M P-state sets, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`Flags`

[in] A set of input flags. No flag bits are currently defined for this member, which is always zero.

`Unit`

[out] A PEP_PERF_STATE_UNIT enumeration value that indicates whether the values for this P-state set are expressed in hertz (frequency units) or bits per second (bandwidth units).

`Type`

[out] A PEP_PERF_STATE_TYPE enumeration value that indicates whether the performance values for this P-state set are expressed as a list of discrete values or as a continuous range of values.

`Discrete`

`Discrete.Count`

`Range`

`Range.Minimum`

`Range.Maximum`

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_SET notification. The **DeviceHandle**, **Component**, **Set**, and **Flags** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent. The **Discrete** member or **Range** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_SET

PEP_DPM_REGISTER_DEVICE

PEP_PERF_STATE_TYPE

PEP_PERF_STATE_UNIT

PEP_QUERY_COMPONENT_PERF_SET_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_SET_NAME** structure contains query information about a set of performance state values (P-state set) for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_SET_NAME {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     Set;
  USHORT    NameSize;
  PWCHAR    Name;
} PEP_QUERY_COMPONENT_PERF_SET_NAME, *PPEP_QUERY_COMPONENT_PERF_SET_NAME;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If this component has M P-state sets, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`NameSize`

[in, out] On input, the size, in bytes, of the buffer pointed to by the **Name** member. If **Name** is NULL, the PEP overwrites the input value of **NameSize** with the buffer size required for the name string.

`Name`

[in] A pointer to an output buffer. The PEP writes the name of the P-state to this buffer. The name is stored as a wide-character, null-terminated string. The **Name** member is NULL if the Windows power management framework (PoFx) needs to determine how large a buffer to allocate for the name string. If **Name** is non-NULL, the buffer must be large enough to contain the entire string, including the terminating null character.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME notification. The

DeviceHandle, **Component**, and **Set** members of the structure contain input values supplied by PoFx when this notification is sent. If the **Name** member is non-NULL, the PEP writes a string to the buffer pointed to by **Name**. If **Name** is NULL, PEP writes the required buffer size to the **NameSize** member.

The string that the PEP writes to the output buffer should contain a descriptive name for the P-state set. This name is intended to make log entries and diagnostic messages easier to understand.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES

PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_COMPONENT_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_STATES** structure contains a list of discrete performance state (P-state) values for the specified P-state set.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_STATES {
  PEPHANDLE       DeviceHandle;
  ULONG           Component;
  ULONG           Set;
  PPEP_PERF_STATE States;
} PEP_QUERY_COMPONENT_PERF_STATES, *PPEP_QUERY_COMPONENT_PERF_STATES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If M is the number of P-state sets for this component, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`States`

[in] A pointer to an output buffer. The PEP writes an array of PEP_PERF_STATE structures to this buffer. Each array element describes one P-state in the P-state set specified by the **Set** member. The Windows power management framework (PoFx) allocated this buffer, which is guaranteed to be large enough to contain an array of the length that the PEP previously wrote to the **Discrete.Count** member of the PEP_QUERY_COMPONENT_PERF_SET structure in response to a PEP_DPM_QUERY_COMPONENT_PERF_SET notification.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_STATES notification. All four members of this structure contain input values that are supplied by PoFx when this notification is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_STATES

PEP_DPM_REGISTER_DEVICE

PEP_PERF_STATE

PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_CURRENT_COMPONENT_PERF_STATE** structure contains information about the current P-state in the specified P-state set.

Syntax

```
typedef struct _PEP_QUERY_CURRENT_COMPONENT_PERF_STATE {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PEP_QUERY_CURRENT_COMPONENT_PERF_STATE, *PPEP_QUERY_CURRENT_COMPONENT_PERF_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If M is the number of P-state sets for this component, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`StateIndex`

`StateValue`

Remarks

This structure is used by the PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE notification. The **DeviceHandle**, **Component**, and **Set** members contain input values supplied by the Windows power management framework (PoFx) when this notification is sent. The **StateIndex** member or **StateValue** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_SOC_SUBSYSTEM structure

1/30/2019 • 3 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM** structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM {
  ULONG          PlatformIdleStateIndex;
  ULONG          SubsystemIndex;
  PVOID          SubsystemHandle;
  UNICODE_STRING ParentName;
  UNICODE_STRING SubsystemName;
  ULONG          MetadataCount;
  ULONG          Flags;
} PEP_QUERY_SOC_SUBSYSTEM, *PPEP_QUERY_SOC_SUBSYSTEM;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index that is being queried by the kernel.

`SubsystemIndex`

[in] The subsystem index, which is a unique index assigned by the OS to each subsystem within the context of a given **PlatformIdleStateIndex**.

The kernel initializes this value to zero and increments it for each subsequent subsystem static info notification. A value of 0 means it's the first such notification for the given **PlatformIdleStateIndex**. This field is intended to give the PEP context between instances of this notification. It is the only field whose input value varies across instances of this notification for a given **PlatformIdleStateIndex** and a PEP might use this value to look up a subsystem.

The PEP can ignore this value.

`SubsystemHandle`

[out] A context pointer that the PEP can optionally assign a value to. It will receive the pointer on subsequent notifications for this particular subsystem. In subsequent notifications, a PEP uses **PlatformIdleStateIndex** along with **SubsystemName** and/or **SubsystemHandle** to look up a particular subsystem's accounting data.

By default the value if this member is zero. This value is ignored by the OS and used only in notifications.

The PEP is not required to fill this member.

`ParentName`

[in/out] A buffer for holding the parent name of the corresponding subsystem. Each subsystem has a parent and subsystems with the same value for **ParentName** are considered siblings. Subsystem sibling hierarchies are optional. In the absence of any hierarchy, all subsystems are top level subsystems and specify a common value for **ParentName**. To indicate a hierarchy, top level subsystems specify a common **ParentName** while each non-top-level subsystem specifies its parent subsystem's **SubsystemName** for **ParentName**.

Prior to entry, the kernel will:

- Pre-allocate 64 **WCHARs** for **ParentName.Buffer[]**
- Zero the buffer
- Set **ParentName.MaximumLength** to the size of the buffer in bytes, and
- Set **ParentName.Length** = **0**

Prior to exit, the PEP must copy a null-terminated string into **ParentName.Buffer** and update **ParentName.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **ParentName.Buffer** to provide the parent name. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the parent name, if necessary, so that it does not exceed the length specified in **ParentName.MaximumLength** (including the terminating **UNICODE_NULL** character).

`SubsystemName`

[in/out] A buffer for holding this subsystem's name. Every subsystem has a subsystem name and **SubsystemName** must be unique among all subsystems within the context of a given platform idle state. A subsystem's **SubsystemName** cannot be the same as **ParentName**.

Prior to entry, the kernel will:

- Pre-allocate 64 **WCHARs** for **SubsystemName.Buffer[]**
- Zero the buffer
- Set **SubsystemName.MaximumLength** to the size of the buffer in bytes, and
- Set **SubsystemName.Length** = **0**

Prior to exit, the PEP must copy a null-terminated string into **SubsystemName.Buffer** and update **SubsystemName.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **SubsystemName.Buffer** to provide the subsystem name. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the subsystem name, if necessary, so that it does not exceed the length specified in **SubsystemName.MaximumLength** (including the terminating **UNICODE_NULL** character).

`MetadataCount`

[out] The number of metadata key/value string pairs tallied by this SoC subsystem.

Metadata is optional. The PEP returns zero if it does not report any metadata for this SoC subsystem.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM notification

PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME** structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME {
  ULONG            PlatformIdleStateIndex;
  PVOID            SubsystemHandle;
  PCUNICODE_STRING SubsystemName;
  ULONG64          BlockingTime;
  ULONG            Flags;
} PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME, *PPEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for the SoC subsystem that the OS is querying.

`SubsystemHandle`

[in] A context pointer that the PEP previously provided on subsystem initialization. The context pointer is optional, so if none was provided then the value will be zero. The PEP is free to ignore this field.

`SubsystemName`

[in] The name of the subsystem whose blocking time is being queried. This name was provided by the PEP on subsystem initialization.

`BlockingTime`

[out] The blocking time tallied for this subsystem in 100 nanosecond units since the previous PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification

PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification

PEP_QUERY_SOC_SUBSYSTEM_COUNT structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_COUNT** structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_COUNT {
  ULONG PlatformIdleStateIndex;
  ULONG SubsystemCount;
  ULONG Flags;
} PEP_QUERY_SOC_SUBSYSTEM_COUNT, *PPEP_QUERY_SOC_SUBSYSTEM_COUNT;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for which the kernel is querying about.

`SubsystemCount`

[out] The number of SoC subsystems tallied by the PEP for the specified platform idle state. The PEP cannot return 0 in this parameter; instead the PEP should return **FALSE** to the PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification that provided this structure.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification

PEP_QUERY_SOC_SUBSYSTEM_METADATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_METADATA** structure is used with the **PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA** notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_METADATA {
  ULONG                       PlatformIdleStateIndex;
  PVOID                       SubsystemHandle;
  PCUNICODE_STRING            SubsystemName;
  ULONG                       Flags;
  ULONG                       MetadataCount;
  PPEP_SOC_SUBSYSTEM_METADATA Metadata[ANYSIZE_ARRAY];
} PEP_QUERY_SOC_SUBSYSTEM_METADATA, *PPEP_QUERY_SOC_SUBSYSTEM_METADATA;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for for the SoC subsystem that the OS is querying.

`SubsystemHandle`

[in] A context pointer that the PEP previously provided on subsystem initialization. The context pointer is optional, so if none was provided then the value will be zero. The PEP is free to ignore this field.

`SubsystemName`

[in] The name of the subsystem whose metadata is being queried.

`Flags`

This member is reserved and should be set to zero.

`MetadataCount`

[in] The number of entries in the **Metadata** array. The PEP previously provided this value as **PEP_QUERY_SOC_SUBSYSTEM.MetadataCount**.

`Metadata`

[in/out] An array of pointers to PEP_SOC_SUBSYSTEM_METADATA structures. Each entry holds one key/value metadata string-pair.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA

PEP_SOC_SUBSYSTEM_METADATA

PEP_REGISTER_COMPONENT_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_COMPONENT_PERF_STATES** structure describes the performance states (P-states) of the specified component.

Syntax

```
typedef struct _PEP_REGISTER_COMPONENT_PERF_STATES {
  PEPHANDLE                DeviceHandle;
  ULONG                    Component;
  ULONGLONG                Flags;
  PPEP_COMPONENT_PERF_INFO PerfStateInfo;
} PEP_REGISTER_COMPONENT_PERF_STATES, *PPEP_REGISTER_COMPONENT_PERF_STATES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`Flags`

[in] A set of input flags. No flag bits are currently defined for this member, which is always zero.

`PerfStateInfo`

[in] A pointer to a PEP_COMPONENT_PERF_INFO structure that contains the performance information for this component's P-states. If this member is non-NULL, it points to P-state information that was supplied by the device driver.

Remarks

This structure is used by the PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification. The values of all members of the **PEP_REGISTER_COMPONENT_PERF_STATES** structure are supplied by the Windows power management framework (PoFx) when the notification is sent. The PEP must not write to this structure.

If the **PerfStateInfo** member is not NULL, the device driver has provided P-state information and requires the PEP to use it.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_COMPONENT_PERF_STATES

PEP_DPM_REGISTER_DEVICE

PEP_REGISTER_CRASHDUMP_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_CRASHDUMP_DEVICE** structure provides a callback routine to turn on a crash-dump device.

Syntax

```
typedef struct _PEP_REGISTER_CRASHDUMP_DEVICE {
  PPEPCALLBACKPOWERONCRASHDUMPDEVICE PowerOnDumpDeviceCallback;
  PEPHANDLE                          DeviceHandle;
} PEP_REGISTER_CRASHDUMP_DEVICE, *PPEP_REGISTER_CRASHDUMP_DEVICE;
```

Members

`PowerOnDumpDeviceCallback`

[out] A pointer to a PowerOnDumpDeviceCallback callback routine that is implemented by the platform extension plug-in (PEP). This routine handles requests from the Windows kernel to turn on the crash-dump device so that a crash dump can be saved. For more information, see Kernel-Mode Dump Files.

`DeviceHandle`

[in] A PEPHANDLE value that identifies the crash-dump device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_REGISTER_CRASHDUMP_DEVICE notification. The **DeviceHandle** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **PowerOnDumpDeviceCallbackmember** contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_CRASHDUMP_DEVICE

PEP_DPM_REGISTER_DEVICE

PowerOnDumpDeviceCallback

PEP_REGISTER_DEBUGGER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_DEBUGGER** structure identifies a registered device that is a core system resource that provides debugger transport.

Syntax

```
typedef struct _PEP_REGISTER_DEBUGGER {
  PEPHANDLE DeviceHandle;
} PEP_REGISTER_DEBUGGER, *PPEP_REGISTER_DEBUGGER;
```

Members

`DeviceHandle`

[in] A **PEPHANDLE** value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_REGISTER_DEBUGGER notification. The **DeviceHandle** member contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEBUGGER

PEP_DPM_REGISTER_DEVICE

PEP_REGISTER_DEVICE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_DEVICE_V2** structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_REGISTER_DEVICE_V2 {
  PCUNICODE_STRING           DeviceId;
  POHANDLE                   KernelHandle;
  PPEP_DEVICE_REGISTER_V2    Register;
  PEPHANDLE                  DeviceHandle;
  PEP_DEVICE_ACCEPTANCE_TYPE DeviceAccepted;
} PEP_REGISTER_DEVICE_V2, *PPEP_REGISTER_DEVICE_V2;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`KernelHandle`

[in] A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Register`

[in] A pointer to a PEP_DEVICE_REGISTER_V2 structure that describes the power management attributes of all the components in the device. For more information, see Remarks.

`DeviceHandle`

[out] A PEPHANDLE value that the PEP creates to identify this device. PoFx will use this handle to identify the device in future device power management (DPM) notifications.

`DeviceAccepted`

[out] A PEP_DEVICE_ACCEPTANCE_TYPE enumeration value that indicates whether the PEP claims ownership of the device. The PEP that claims ownership is responsible for handling DPM notifications for the device.

Remarks

This structure is used by the PEP_DPM_REGISTER_DEVICE notification. The first three members of this structure contain input values supplied by PoFx. The last two members contain output values that the PEP writes to the structure in response to this notification.

The **Register** member contains a pointer to an input buffer allocated by PoFx. PoFx writes the PEP_DEVICE_REGISTER_V2 structure and associated data to this structure before sending the PEP_DPM_REGISTER_DEVICE notification to the PEP. The contents of this buffer remain valid only until the PEP

finishes handling the notification and returns from the AcceptDeviceNotification callback.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

AcceptDeviceNotification

PEP_DEVICE_ACCEPTANCE_TYPE

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

UNICODE_STRING

PEP_REQUEST_COMPONENT_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REQUEST_COMPONENT_PERF_STATE** structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_REQUEST_COMPONENT_PERF_STATE {
  PEPHANDLE                          DeviceHandle;
  ULONG                              Component;
  BOOLEAN                            Completed;
  BOOLEAN                            Succeeded;
  ULONG                              PerfRequestsCount;
  PPEP_COMPONENT_PERF_STATE_REQUEST PerfRequests;
} PEP_REQUEST_COMPONENT_PERF_STATE, *PPEP_REQUEST_COMPONENT_PERF_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Completed`

[out] Whether the PEP has completed the requested P-state changes. Set to TRUE to indicate that the P-state changes completed synchronously. Set to FALSE to indicate that the requested P-state changes are pending and will be completed asynchronously by a PEP worker thread. For more information, see Remarks.

`Succeeded`

[out] Whether the requested P-state changes successfully completed. Set to TRUE if the PEP successfully completed all P-state changes requested in the **PerfRequests** array. Set to FALSE if the PEP was unable to perform all the requested P-state changes, in which case the P-states in the hardware were left unchanged. The **Succeeded** member is used only for *synchronous* P-state changes. If the PEP sets **Completed** to FALSE, PoFx ignores the value in the **Succeeded** member.

`PerfRequestsCount`

[in] The number of P-state changes requested in the array pointed to by the **PerfRequests** member.

`PerfRequests`

[in] A pointer to an array of PEP_COMPONENT_PERF_STATE_REQUEST structures. Each element in the array specifies a new performance level to assign to a P-state set. The number of array elements is specified by the

PerfRequestsCount member.

Remarks

This structure is used by the PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification. The **DeviceHandle**, **Component**, **PerfRequestsCount**, and **PerfRequests** members of the structure contain input values that are supplied by PoFx when this notification is sent. The **Completed** and **Succeeded** members contain output values that the PEP writes to the structure in response to the notification.

The PEP can complete the requested P-state changes either synchronously or asynchronously. The PEP can set the **Completed** member to TRUE to indicate that all the requests in the **PEP_DPM_REQUEST_COMPONENT_PERF_STATE** notification have been completed synchronously—that is, before the return from the PEP's AcceptDeviceNotification callback routine. Or, the PEP can set **Completed** to FALSE to indicate that the requests will be completed asynchronously. To complete the requests asynchronously, the PEP should call the RequestWorker routine to inform PoFx that the PEP has a work request to submit, and PoFx will respond by sending a PEP_DPM_WORK notification to the PEP.

PoFx writes the requested P-state changes to the **PerfRequests** array before the **PEP_DPM_REQUEST_COMPONENT_PERF_STATE** notification is sent. If the PEP handles the requested P-state changes *synchronously*, the contents of the **PerfRequests** array remain valid only until the return from the *AcceptDeviceNotification* callback routine. If the requests are completed *asynchronously*, the array contents remain valid until the PEP calls the CompleteWork routine to notify PoFx that the PEP has completed the pending requests.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

AcceptDeviceNotification

PEP_COMPONENT_PERF_STATE_REQUEST

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_DPM_WORK

RequestWorker

PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING** structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.

Syntax

```
typedef struct _PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING {
  ULONG PlatformIdleStateIndex;
  ULONG Flags;
} PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING, *PPEP_RESET_SOC_SUBSYSTEM_ACCOUNTING;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for the subsystem accounting to be reset.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification

PEP_SOC_SUBSYSTEM_METADATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_SOC_SUBSYSTEM_METADATA** structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_SOC_SUBSYSTEM_METADATA {
  UNICODE_STRING Key;
  UNICODE_STRING Value;
} PEP_SOC_SUBSYSTEM_METADATA, *PPEP_SOC_SUBSYSTEM_METADATA;
```

Members

`Key`

[in/out] A buffer for the PEP to write the key portion of the metadata string-pair. **Key** must be unique among all **Key** values reported by this subsystem.

Prior to entry into the notification callback routine, the kernel will:

- Pre-allocate 64 **WCHARs** for **Key.Buffer[]**
- Zero the buffer
- Set **Key.MaximumLength** to the size of the buffer in bytes, and
- Set **Key.Length** = **0**

Prior to exit from the callback routine, the PEP must copy a null-terminated string into **Key.Buffer** and update **Key.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **Key.Buffer** to provide the key. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the key string, if necessary, so that it does not exceed the length specified in **Key.MaximumLength** (including the terminating **UNICODE_NULL** character).

`Value`

[in/out] A buffer for the PEP to write the value portion of the metadata string-pair.

Prior to entry into the notification callback routine, the kernel will:

- Pre-allocate 64 **WCHARs** for **Value.Buffer[]**
- Zero the buffer
- Set **Value.MaximumLength** to the size of the buffer in bytes, and
- Set **Value.Length** = **0**

Prior to exit from the callback routine, the PEP must copy a null-terminated string into **Value.Buffer** and update **Value.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **Value.Buffer** to provide the metadata value. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the value string, if necessary, so that it does not exceed the length specified in **Value.MaximumLength** (including the terminating **UNICODE_NULL** character).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA

PEP_QUERY_SOC_SUBSYSTEM_METADATA

PEP_SYSTEM_LATENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_SYSTEM_LATENCY** structure specifies the new value for the system latency tolerance.

Syntax

```
typedef struct _PEP_SYSTEM_LATENCY {
  ULONGLONG Latency;
} PEP_SYSTEM_LATENCY, *PPEP_SYSTEM_LATENCY;
```

Members

`Latency`

[in] The overall system latency tolerance, in 100-nanosecond units. This member specifies the maximum latency that the operating system can tolerate in the time required to move a component from a low-power F*x* state to F0.

Remarks

This structure is used by the PEP_DPM_SYSTEM_LATENCY_UPDATE notification. The **Latency** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_SYSTEM_LATENCY_UPDATE

PEP_UNMASKED_INTERRUPT_FLAGS union

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_UNMASKED_INTERRUPT_FLAGS** union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.

Syntax

```
typedef union _PEP_UNMASKED_INTERRUPT_FLAGS {
  struct {
    USHORT SecondaryInterrupt : 1;
    USHORT Reserved : 15;
  };
  USHORT AsUSHORT;
} PEP_UNMASKED_INTERRUPT_FLAGS, *PPEP_UNMASKED_INTERRUPT_FLAGS;
```

Members

`SecondaryInterrupt`

`Reserved`

`AsUSHORT`

A USHORT value that contains all of the unmasked interrupt flags.

Remarks

The **Flags** member of the PEP_UNMASKED_INTERRUPT_INFORMATION structure is a **PEP_UNMASKED_INTERRUPT_FLAGS** union.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_UNMASKED_INTERRUPT_INFORMATION

PEP_UNMASKED_INTERRUPT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_UNMASKED_INTERRUPT_INFORMATION** structure contains information about an interrupt source.

Syntax

```
typedef struct _PEP_UNMASKED_INTERRUPT_INFORMATION {
  USHORT                       Version;
  USHORT                       Size;
  PEP_UNMASKED_INTERRUPT_FLAGS Flags;
  KINTERRUPT_MODE              Mode;
  KINTERRUPT_POLARITY          Polarity;
  ULONG                        Gsiv;
  USHORT                       PinNumber;
  PEPHANDLE                    DeviceHandle;
} PEP_UNMASKED_INTERRUPT_INFORMATION, *PPEP_UNMASKED_INTERRUPT_INFORMATION;
```

Members

`Version`

The version of this structure.

`Size`

The size, in bytes, of this structure.

`Flags`

A PEP_UNMASKED_INTERRUPT_FLAGS union that indicates whether the interrupt is a primary or secondary interrupt. For more information, see Primary and Secondary Interrupts.

`Mode`

A KINTERRUPT_MODE enumeration value. This member indicates whether the interrupt is edge-triggered or level-triggered.

`Polarity`

A KINTERRUPT_POLARITY enumeration value. This member indicates which edge or level of the interrupt signal triggers the interrupt.

`Gsiv`

The global system interrupt vector (GSIV) number that identifies this interrupt. The ACPI firmware assigns GSIV numbers to all primary interrupt lines. For secondary (GPIO) interrupt lines, the GSIV number is dynamically assigned by the operating system.

`PinNumber`

For secondary interrupt sources, this member identifies the number of the pin on the general-purpose I/O (GPIO) controller that is connected to the interrupt signal line from the interrupting device. For primary interrupt sources,

this member is undefined.

If a GPIO controller has N GPIO pins, the pins are numbered 0 to N−1. One or more of these GPIO pins might be configured as interrupt inputs.

`DeviceHandle`

For secondary interrupt sources, this member contains the PEP device handle for the GPIO controller that is the source for this interrupt. For primary interrupt sources, this field is undefined.

Remarks

This structure is used by the EnumerateInterruptSource callback routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

EnumerateInterruptSource

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_UNMASKED_INTERRUPT_FLAGS

PEP_UNREGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_UNREGISTER_DEVICE** structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_UNREGISTER_DEVICE {
  PEPHANDLE DeviceHandle;
} PEP_UNREGISTER_DEVICE, *PPEP_UNREGISTER_DEVICE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_UNREGISTER_DEVICE notification. The **DeviceHandle** member contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_DPM_UNREGISTER_DEVICE

PEP_WORK structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK** structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_WORK {
  PPEP_WORK_INFORMATION WorkInformation;
  BOOLEAN               NeedWork;
} PEP_WORK, *PPEP_WORK;
```

Members

`WorkInformation`

[out] A pointer to a PEP-allocated PEP_WORK_INFORMATION structure that describes the work that the PEP is requesting. If **NeedWork** is TRUE, **WorkInformation** must point to a valid **PEP_WORK_INFORMATION** structure. If **NeedWork** is FALSE, **WorkInformation** must be NULL.

`NeedWork`

[out] Whether the PEP has a work request to submit. Set this member to TRUE if the PEP has a work request, or to FALSE if the PEP has no work to request.

Remarks

This structure is used by the PEP_DPM_WORK notification. Both members of the structure contain values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_WORK

PEP_WORK_INFORMATION

PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE** structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE {
  POHANDLE              DeviceHandle;
  ULONG                 CompletionFlags;
  NTSTATUS              MethodStatus;
  PVOID                 CompletionContext;
  SIZE_T                OutputArgumentSize;
  PACPI_METHOD_ARGUMENT OutputArguments;
} PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE, *PPEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE;
```

Members

`DeviceHandle`

A POHANDLE value that represents the registration of the device with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification in which the PEP elected to provide ACPI services for the device.

`CompletionFlags`

A set of completion flags. No flags are currently defined for this member, which is always set to PEP_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE_FLAG_NONE (0x0).

`MethodStatus`

`CompletionContext`

A pointer to a completion context value. This is the same pointer that PoFx passed to the PEP in the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification that initiated the asynchronous evaluation of the ACPI control method.

`OutputArgumentSize`

The size, in bytes, of the buffer pointed to by the **OutputArguments** member.

`OutputArguments`

A pointer to a buffer to which the PEP has written the result of evaluating the ACPI control method. This buffer starts with a ACPI_METHOD_ARGUMENT structure, which might be followed by additional output data. PoFx allocated this buffer and passed the buffer pointer to the PEP in the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification that initiated the asynchronous evaluation of the ACPI control method.

Remarks

The **ControlMethodComplete** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE** structure.

A **PEP_WORK_INFORMATION** structure is included in the input parameters to the CompleteWork routine. The PEP writes the control method results to the **ControlMethodComplete** member of this structure before calling the routine. Evaluation of the control method was initiated by a previous PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_ACPI_NOTIFY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_ACPI_NOTIFY** structure contains the ACPI Notify code for a device that has generated a hardware event.

Syntax

```
typedef struct _PEP_WORK_ACPI_NOTIFY {
  POHANDLE DeviceHandle;
  ULONG    NotifyCode;
} PEP_WORK_ACPI_NOTIFY, *PPEP_WORK_ACPI_NOTIFY;
```

Members

`DeviceHandle`

A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from the Windows power management framework (PoFx) during the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification in which the PEP elected to provide ACPI services for the device.

`NotifyCode`

The Notify code. For a list of Notify codes and their meanings, see section 5.6.6, "Device Object Notifications", of the ACPI 5.0 specification.

Remarks

The **AcpiNotify** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_ACPI_NOTIFY** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_ACTIVE_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_ACTIVE_COMPLETE** structure identifies a component that is now in the *active* condition.

Syntax

```
typedef struct _PEP_WORK_ACTIVE_COMPLETE {
  POHANDLE DeviceHandle;
  ULONG    Component;
} PEP_WORK_ACTIVE_COMPLETE, *PPEP_WORK_ACTIVE_COMPLETE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Remarks

The **ActiveComplete** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_ACTIVE_COMPLETE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_COMPLETE_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_COMPLETE_IDLE_STATE** structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new F*x* power state.

Syntax

```
typedef struct _PEP_WORK_COMPLETE_IDLE_STATE {
  POHANDLE DeviceHandle;
  ULONG    Component;
} PEP_WORK_COMPLETE_IDLE_STATE, *PPEP_WORK_COMPLETE_IDLE_STATE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Remarks

The **CompleteIdleState** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_COMPLETE_IDLE_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_COMPLETE_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_COMPLETE_PERF_STATE** structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.

Syntax

```
typedef struct _PEP_WORK_COMPLETE_PERF_STATE {
  POHANDLE DeviceHandle;
  ULONG    Component;
  BOOLEAN  Succeeded;
} PEP_WORK_COMPLETE_PERF_STATE, *PPEP_WORK_COMPLETE_PERF_STATE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). The PEP received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Succeeded`

Whether the requested P-state changes successfully completed. Set to TRUE if the PEP successfully completed all P-state changes requested in a previous PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification. Set to FALSE if the PEP was unable to perform all the requested P-state changes, in which case the P-states in the hardware were left unchanged.

Remarks

The **CompletePerfState** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_COMPLETE_PERF_STATE** structure. If PoFx sends a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification to request performance level changes to one or more P-state sets, and the PEP chooses to handle this request asynchronously, the PEP uses a **PEP_WORK_COMPLETE_PERF_STATE** structure to describe the completion status of the request.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_WORK_INFORMATION

PEP_WORK_DEVICE_IDLE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_DEVICE_IDLE** structure indicates whether to ignore the idle time-out for the specified device.

Syntax

```
typedef struct _PEP_WORK_DEVICE_IDLE {
  POHANDLE DeviceHandle;
  BOOLEAN  IgnoreIdleTimeout;
} PEP_WORK_DEVICE_IDLE, *PPEP_WORK_DEVICE_IDLE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`IgnoreIdleTimeout`

Whether to ignore the idle time-out. Set to TRUE to tell the operating system to ignore the idle time-out for this device. Set to FALSE to tell the operating system to call the device driver's DevicePowerNotRequiredCallback callback routine after the device remains idle for the designated time-out interval. For more information, see PoFxSetDeviceIdleTimeout.

Remarks

The **DeviceIdle** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_DEVICE_IDLE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PoFxSetDeviceIdleTimeout

PEP_WORK_DEVICE_POWER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_DEVICE_POWER** structure describes the new power requirements for the specified device.

Syntax

```
typedef struct _PEP_WORK_DEVICE_POWER {
  POHANDLE DeviceHandle;
  BOOLEAN  PowerRequired;
} PEP_WORK_DEVICE_POWER, *PPEP_WORK_DEVICE_POWER;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`PowerRequired`

Whether the device requires power. Set to TRUE if the device requires power and the device driver must bring the device back to the D0 power state. Set to FALSE if the device is not required to stay turned on and the driver is permitted to transition the device from D0 to a low-power state.

Remarks

The **DevicePower** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_DEVICE_POWER** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_IDLE_STATE** structure contains a request to transition a component to an F*x* power state.

Syntax

```
typedef struct _PEP_WORK_IDLE_STATE {
  POHANDLE DeviceHandle;
  ULONG    Component;
  ULONG    State;
} PEP_WORK_IDLE_STATE, *PPEP_WORK_IDLE_STATE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`State`

The target F*x* power state to transition the component to. If this parameter is 0, the target state is F0; if this parameter is 1, the target state is F1; and so on. For more information about F*x* power states, see Component-Level Power Management.

Remarks

The **IdleState** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_IDLE_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pepfx.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_INFORMATION structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_INFORMATION** structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_WORK_INFORMATION {
  PEP_WORK_TYPE WorkType;
  union {
    PEP_WORK_POWER_CONTROL                          PowerControl;
    PEP_WORK_COMPLETE_IDLE_STATE                    CompleteIdleState;
    PEP_WORK_COMPLETE_PERF_STATE                    CompletePerfState;
    PEP_WORK_ACPI_NOTIFY                            AcpiNotify;
    PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE ControlMethodComplete;
  };
} PEP_WORK_INFORMATION, *PPEP_WORK_INFORMATION;
```

Members

`WorkType`

A PEP_WORK_TYPE enumeration value. This member indicates the type of work requested by the PEP, which also determines the type of structure that is contained in the unnamed union in the **PEP_WORK_INFORMATION** structure.

`PowerControl`

`CompleteIdleState`

`CompletePerfState`

`AcpiNotify`

`ControlMethodComplete`

Remarks

The **WorkInformation** member of the PEP_WORK structure is a pointer to a **PEP_WORK_INFORMATION** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_WORK

PEP_WORK

PEP_WORK_ACPI_NOTIFY

PEP_WORK_ACTIVE_COMPLETE

PEP_WORK_COMPLETE_IDLE_STATE

PEP_WORK_COMPLETE_PERF_STATE

PEP_WORK_DEVICE_IDLE

PEP_WORK_DEVICE_POWER

PEP_WORK_IDLE_STATE

PEP_WORK_POWER_CONTROL

PEP_WORK_TYPE

PEP_WORK_POWER_CONTROL structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_POWER_CONTROL** structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.

Syntax

```
typedef struct _PEP_WORK_POWER_CONTROL {
  POHANDLE DeviceHandle;
  LPCGUID  PowerControlCode;
  PVOID    RequestContext;
  PVOID    InBuffer;
  SIZE_T   InBufferSize;
  PVOID    OutBuffer;
  SIZE_T   OutBufferSize;
} PEP_WORK_POWER_CONTROL, *PPEP_WORK_POWER_CONTROL;
```

Members

`DeviceHandle`

A POHANDLE value that identifies the processor. The PEP received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation to perform.

`RequestContext`

A pointer to a PEP-defined context value. The PEP might use this value to uniquely identify a power control request message (in case the PEP issues multiple requests with the same control code to the same device).

`InBuffer`

[in] A pointer to a driver-allocated input buffer that contains the input parameters for this power control operation.

`InBufferSize`

[in] The size in bytes of the buffer pointed to by **InBuffer**.

`OutBuffer`

[in] A pointer to a driver-allocated output buffer to which the PEP writes the results of this power control operation.

`OutBufferSize`

[in] The size in bytes of the buffer pointed to by **OutBuffer**.

Remarks

The **PowerControl** member of the PEP_WORK_INFORMATION structure is a pointer to a **PEP_WORK_POWER_CONTROL** structure.

The PEP issues a power control request from a worker thread. The Windows power management framework (PoFx) forwards this request by calling the processor driver's PowerControlCallback routine, if the driver implements this routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

GUID

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PoFxRegisterDevice

PowerControlCallback

PEP_WORK_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_TYPE** enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.

Syntax

```
typedef enum _PEP_WORK_TYPE {
  PepWorkRequestPowerControl,
  PepWorkCompleteIdleState,
  PepWorkCompletePerfState,
  PepWorkAcpiNotify,
  PepWorkAcpiEvaluateControlMethodComplete,
  PepWorkMax
} PEP_WORK_TYPE, *PPEP_WORK_TYPE;
```

Constants

PepWorkRequestPowerControl	A request for the device driver to perform a custom power-control operation that uses a device-specific context that the PEP provides for the operation. The driver handles this request in its PowerControlCallback routine.
PepWorkCompleteIdleState	A notification to PoFx that the PEP has asynchronously completed the transition of a component to an idle state. PoFx previously initiated this transition by sending a PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE notification to the PEP.
PepWorkCompletePerfState	A notification to PoFx that the PEP has asynchronously completed the transition of a component to a P-state. PoFx previously initiated this transition by sending a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification to the PEP.
PepWorkAcpiNotify	An ACPI Notify code to describe to PoFx a hardware event that the specified device has generated.
PepWorkAcpiEvaluateControlMethodComplete	A notification to PoFx that the PEP has asynchronously finished evaluating an ACPI control method. PoFx previously initiated the evaluation of this method by sending a PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification to the PEP.
PepWorkMax	Reserved for use by operating system.

Remarks

This enumeration is used by the PEP_WORK_INFORMATION structure to describe the type of work item that the PEP is requesting.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_WORK_INFORMATION

PowerControlCallback

PO_FX_CORE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_CORE_DEVICE** structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.

Syntax

```
typedef struct _PO_FX_CORE_DEVICE {
  ULONG                                        Version;
  ULONG                                        ComponentCount;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK    ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK      ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK ComponentCriticalTransitionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK          ComponentIdleStateCallback;
  PVOID                                        DeviceContext;
  PO_FX_COMPONENT                              Components[ANYSIZE_ARRAY];
} PO_FX_CORE_DEVICE, *PPO_FX_CORE_DEVICE;
```

Members

`Version`

The version number of this structure.

`ComponentCount`

The number of elements in the **Components** array at the end of this structure. This count includes all elements of the array, and includes both the first element, which is a member of this structure, and any additional elements that follow the end of this structure.

`ComponentActiveConditionCallback`

A pointer to a ComponentActiveConditionCallback callback routine. PoFx calls this routine to notify the driver that the specified component completed a transition from the idle condition to the active condition.

`ComponentIdleConditionCallback`

A pointer to a ComponentIdleConditionCallback callback routine. PoFx calls this routine to notify the driver that the specified component completed a transition from the active condition to the idle condition.

`ComponentCriticalTransitionCallback`

A pointer to a ComponentCriticalTransitionCallback callback routine. PoFx calls this routine to notify the driver that the specified component is about to make a transition between the idle condition and the active condition.

`ComponentIdleStateCallback`

A pointer to a ComponentIdleStateCallback callback routine. PoFx calls this routine to notify the driver of a pending change to the F*x* power state of the specified component.

`DeviceContext`

A pointer to a PEP-defined device context. PoFx supplies this pointer as the first parameter in calls to the *ComponentActiveConditionCallback*, *ComponentIdleConditionCallback*, *ComponentCriticalTransitionCallback*, and

ComponentIdleStateCallback routines. This context is opaque to PoFx.

`Components`

The first element in an array of PO_FX_COMPONENT structures. Each element in the array describes the power attributes of a component in the device. If the array contains more than one element, the additional elements immediately follow the end of the **PO_FX_CORE_DEVICE** structure. The number of elements in the array is specified in the **ComponentCount** member of the **PO_FX_CORE_DEVICE** structure.

Remarks

This structure is used by the PoFxRegisterCoreDevice routine.

A *core system resource* is a hardware device, such as a timer or an interrupt controller, that is managed by the Windows hardware abstraction layer (HAL).

For more information about component transitions between the active condition and the idle condition, see Component-Level Power Management.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pep_x.h (include Pep_x.h)

See Also

ComponentActiveConditionCallback

ComponentCriticalTransitionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

PO_FX_COMPONENT

PoFxRegisterCoreDevice

PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.

Syntax

```
void PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE(
  BOOLEAN            ResourceUsage,
  UCHAR              Decode,
  BOOLEAN            IsMinFixed,
  BOOLEAN            IsMaxFixed,
  UCHAR              ISARanges,
  ULONGLONG          AddressGranularity,
  ULONGLONG          AddressMinimum,
  ULONGLONG          AddressMaximum,
  ULONGLONG          AddressTranslation,
  ULONGLONG          RangeLength,
  ULONGLONG          TypeSpecificAttributes,
  PUNICODE_STRING    DescriptorName,
  BOOLEAN            TranslationTypeNonStatic,
  BOOLEAN            TanslationSparseDensity,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ResourceUsage`

This parameter is copied into the **GeneralFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`Decode`

When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only.

When not set, indicates that this bridge positively decodes this address.

`IsMinFixed`

When set, indicates that the minimum address is fixed.

`IsMaxFixed`

When set, indicates that the maximum address is fixed.

`ISARanges`

This parameter is copied into the **TypeSpecificFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`AddressGranularity`

A bit mask indicating which bits have been decoded.

`AddressMinimum`

For bridges that translate addresses, this indicates the minimum starting address on the secondary side of the bridge.

`AddressMaximum`

For bridges that translate addresses, this indicates the maximum starting address on the secondary side of the bridge.

`AddressTranslation`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`RangeLength`

The length of the address range.

`TypeSpecificAttributes`

The type-specific attributes for this resource.

`DescriptorName`

The name of the resource descriptor.

`TranslationTypeNonStatic`

When true, indicates that the resource uses type translation. Otherwise, it uses type-static translation.

`TanslationSparseDensity`

When false, indicates that this is a dense translation. Otherwise, it is sparse.

`Resource`

This is cast to *PEP_ACPI_EXTENDED_ADDRESS.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.

Syntax

```
void PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE(
  BOOLEAN            ResourceUsage,
  UCHAR              Decode,
  BOOLEAN            IsMinFixed,
  BOOLEAN            IsMaxFixed,
  UCHAR              Cacheable,
  BOOLEAN            ReadWrite,
  ULONGLONG          AddressGranularity,
  ULONGLONG          AddressMinimum,
  ULONGLONG          AddressMaximum,
  ULONGLONG          AddressTranslation,
  ULONGLONG          RangeLength,
  ULONGLONG          TypeSpecificAttributes,
  PUNICODE_STRING    DescriptorName,
  UCHAR              MemoryRangeType,
  BOOLEAN            TranslationTypeNonStatic,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ResourceUsage`

This parameter is copied into the **GeneralFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`Decode`

When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only.

When not set, indicates that this bridge positively decodes this address.

`IsMinFixed`

When set, indicates that the minimum address is fixed.

`IsMaxFixed`

When set, indicates that the maximum address is fixed.

`Cacheable`

The caching flag for the resource.

VALUE	MEANING

0	Indicates the memory is non-cacheable.
1	Indicates the memory is cacheable.
2	Indicates the memory is cacheable and supports write combining.
3	The memory is cacheable and prefetchable.

`ReadWrite`

When true, indicates that the resource is available for read/write access. Otherwise, it's read-only.

`AddressGranularity`

A bit mask indicating which bits have been decoded.

`AddressMinimum`

For bridges that translate addresses, this indicates the minimum starting address on the secondary side of the bridge.

`AddressMaximum`

For bridges that translate addresses, this indicates the maximum starting address on the secondary side of the bridge.

`AddressTranslation`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`RangeLength`

The length of the address range.

`TypeSpecificAttributes`

The type-specific attributes for this resource.

`DescriptorName`

The name of the resource descriptor.

`MemoryRangeType`

This parameter identifies the type of memory range provided by this resource.

VALUE	MEANING
0	Address range memory: This range is available RAM usable by the operating system.
1	Address range reserved: This range of addresses is in use or reserved by the system and is not to be included in the allocatable memory pool of the operating system's memory manager.
2	Address range ACPI: ACPI Reclaim Memory. This range is available RAM usable by the OS after it reads the ACPI tables.

3	Address Range NVS: ACPI NVS Memory. This range of addresses is in use or reserved by the system and must not be used by the operating system. This range is required to be saved and restored across an NVS sleep.

`TranslationTypeNonStatic`

When true, indicates that the resource uses type translation. Otherwise, it uses type-static translation.

`Resource`

This is cast to *PEP_ACPI_EXTENDED_ADDRESS.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE(
  KINTERRUPT_MODE      InterruptType,
  KINTERRUPT_POLARITY  LevelType,
  BOOLEAN              Shareable,
  BOOLEAN              CanWake,
  GPIO_PIN_CONFIG_TYPE PinConfig,
  USHORT               DebounceTimeout,
  UCHAR                ResourceSourceIndex,
  PUNICODE_STRING      ResourceSourceName,
  BOOLEAN              ResourceUsage,
  PUCHAR               VendorData,
  USHORT               VendorDataLength,
  PUSHORT              PinTable,
  UCHAR                PinCount,
  PPEP_ACPI_RESOURCE   Resource
);
```

Parameters

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`LevelType`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`Shareable`

Indicates if the device can be shared.

`CanWake`

Indicates if the device can be woken from a low-power state.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceSourceName`

This parameter should always be "ResourceConsumer."

`ResourceUsage`

Indicates if this device is in use.

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the buffer in the *VendorData* partameter.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_GPIO_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE(
  BOOLEAN                     Shareable,
  BOOLEAN                     CanWake,
  GPIO_PIN_CONFIG_TYPE        PinConfig,
  USHORT                      DebounceTimeout,
  USHORT                      DriveStrength,
  GPIO_PIN_IORESTRICTION_TYPE IoRestriction,
  UCHAR                       ResourceSourceIndex,
  PUNICODE_STRING             ResourceSourceName,
  BOOLEAN                     ResourceUsage,
  PUCHAR                      VendorData,
  USHORT                      VendorDataLength,
  PUSHORT                     PinTable,
  USHORT                      PinCount,
  PPEP_ACPI_RESOURCE          Resource
);
```

Parameters

`Shareable`

Indicates if the device can be shared.

`CanWake`

Indicates if the device can be woken from a low-power state.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`DriveStrength`

`IoRestriction`

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceSourceName`

This parameter should always be "ResourceConsumer."

`ResourceUsage`

Indicates if this device is in use.

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the buffer in the *VendorData* partameter.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_GPIO_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE(
  BOOLEAN             ResourceUsage,
  KINTERRUPT_MODE     EdgeLevel,
  KINTERRUPT_POLARITY InterruptLevel,
  BOOLEAN             ShareType,
  BOOLEAN             Wake,
  PULONG              PinTable,
  UCHAR               PinCount,
  PPEP_ACPI_RESOURCE  Resource
);
```

Parameters

`ResourceUsage`

Indicates if this device is in use.

`EdgeLevel`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptLevel`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`ShareType`

Indicates if the device can be shared.

`Wake`

Indicates if the device can be woken from a low-power state.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_INTERRUPT_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_INTERRUPT_RESOURCE

PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_IOPORT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_IOPORT_RESOURCE(
  UCHAR               Decode,
  USHORT              MinimumAddress,
  USHORT              MaximumAddress,
  UCHAR               Alignment,
  UCHAR               PortLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`Decode`

Specifies how to decode the resource address. If bit 0 is a 1, this indicates that the logical device decodes 16-bit addresses. If bit 0 is 0, this indicates that the logical device only decodes the first 10 bits of the address.

Bits 1 to 7 of this parameter are reserved and must be set to zero.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the IO address assigned.

`PortLength`

Specifies the number of bytes in the IO range.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_IO_MEMORY_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_MEMORY_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_MEMORY_RESOURCE(
  UCHAR              ReadWrite,
  ULONG              MinimumAddress,
  ULONG              MaximumAddress,
  ULONG              Alignment,
  ULONG              MemorySize,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ReadWrite`

If true, indicates that the resource is read/write. Otherwise, it's read-only.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the memory address assigned.

`MemorySize`

Specifies the number of bytes in the memory range.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_IO_MEMORY_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.

Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE(
  USHORT              SlaveAddress,
  BOOLEAN             DeviceInitiated,
  ULONG               ConnectionSpeed,
  BOOLEAN             AddressingMode,
  PUNICODE_STRING     ResourceSource,
  UCHAR               ResourceSourceIndex,
  BOOLEAN             ResourceUsage,
  BOOLEAN             SharedMode,
  PCHAR               VendorData,
  USHORT              VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`SlaveAddress`

The I2C bus address for this connection.

`DeviceInitiated`

If true, indicates that communication over this connection is initiated by the device.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`AddressingMode`

Indicates that this device is in addressing mode.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be set to zero.

`ResourceUsage`

Indicates if the resource is in use.

`SharedMode`

Indicates if the resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_I2C_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_I2C_RESOURCE

PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE(
  USHORT              DeviceSelection,
  UCHAR               DeviceSelectionPolarity,
  UCHAR               WireMode,
  UCHAR               DataBitLength,
  BOOLEAN             SlaveMode,
  ULONG               ConnectionSpeed,
  UCHAR               ClockPolarity,
  UCHAR               ClockPhase,
  PUNICODE_STRING     ResourceSource,
  UCHAR               ResourceSourceIndex,
  BOOLEAN             ResourceUsage,
  BOOLEAN             SharedMode,
  PCHAR               VendorData,
  USHORT              VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`DeviceSelection`

The device selection value. This value is specific to the device and may refer to a chip-select line, GPIO line, or other line selection mechanism.

`DeviceSelectionPolarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`WireMode`

When zero, indicates that this device produces and consumes this resource. Otherwise, this device only consumes this resource.

`DataBitLength`

The size, in bits, of the smallest unit of transfer.

`SlaveMode`

Indicates if the resource is operating in slave mode.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`ClockPolarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`ClockPhase`

The phase of the clock pulse on which to capture data.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be set to zero.

`ResourceUsage`

Indicates if the resource is in use.

`SharedMode`

Indicates if the resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_SPI_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_SPI_RESOURCE

PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE(
  ULONG              BaudRate,
  UCHAR              BitsPerByte,
  UCHAR              StopBits,
  UCHAR              LinesInUse,
  UCHAR              IsBigEndian,
  UCHAR              Parity,
  UCHAR              FlowControl,
  USHORT             RxSize,
  USHORT             TxSize,
  PUNICODE_STRING    ResourceSource,
  UCHAR              ResourceSourceIndex,
  BOOLEAN            ResourceUsage,
  BOOLEAN            SharedMode,
  PCHAR              VendorData,
  USHORT             VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`BaudRate`

Specifies the baud rate of the connection.

`BitsPerByte`

Specifies the number of bits per byte of data.

`StopBits`

Specifies the stop bits used in the connection.

`LinesInUse`

Flag indicating the serial lines that are enabled. A value of 1 in the bit positions indicates that the line is enabled.

BIT	MEANING
0	This bit is reserved and must be set to zero.
1	This bit is reserved and must be set to zero.

2	Data Carrier Detect (DTD)
3	Ring Indicator (RI)
4	Data Set Ready (DSR)
5	Data Terminal Ready (DTR)
6	Clear to Send (CTS)
7	Request to Send (RTS)

`IsBigEndian`

Indicates if the most significant bits of data are in the lowest address.

`Parity`

Specifies the parity of the connection.

VALUE	MEANING
0x0 0	None
0x0 1	Even
0x0 2	Odd
0x0 3	Mark
0x0 4	Space

`FlowControl`

Specifies the type of flow control used by the connection.

`RxSize`

Specifies the maximum receive buffer size, in bytes, that is supported by this connection.

`TxSize`

Specifies the maximum transmit buffer size, in bytes, that is supported by this connection.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceUsage`

Indicates if this resource is in use.

`SharedMode`

Indicates if this resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_UART_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_UART_RESOURCE

PEPCALLBACKNOTIFYACPI callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptAcpiNotification* event callback routine handles ACPI notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYACPI Pepcallbacknotifyacpi;

BOOLEAN Pepcallbacknotifyacpi(
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Notification`

The ACPI notification ID. This parameter specifies the type of ACPI notification that PoFx is sending. For a list of ACPI notification IDs, see ACPI notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various ACPI notification IDs, see ACPI notifications.

Return Value

The *AcceptAcpiNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a device. The **AcceptAcpiNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptAcpiNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptAcpiNotification* routine with PoFx.

The IRQL at which the PEP's *AcceptAcpiNotification* routine is called varies according to the type of notification that is being sent. The notification type is specified by the *Notification* parameter. For more information about the IRQL at which each notification type is sent, see the individual reference pages under ACPI notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.

Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKNOTIFYDPM callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptDeviceNotification* event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYDPM Pepcallbacknotifydpm;

BOOLEAN Pepcallbacknotifydpm(
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Notification`

The DPM notification ID. This parameter specifies the type of DPM notification that PoFx is sending. For a list of DPM notification IDs, see Device power management (DPM) notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various DPM notification IDs, see Device power management (DPM) notifications.

Return Value

The *AcceptDeviceNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a device. The **AcceptDeviceNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptDeviceNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptDeviceNotification* routine with PoFx.

PoFx calls this routine to send a DPM notification to the PEP. In this call, the *Notification* parameter specifies the type of notification, and the *Data* parameter points to a data structure of the type that is associated with this notification. For a list of DPM notification IDs and their associated structure types, see Device power management (DPM) notifications.

DPM notifications inform the PEP about various system, device, or componentÂ–-level events, or pass requests from the device driver to the PEP.

The *AcceptDeviceNotification* routine does not have to accept every request from the device driver—for example, it can ignore idle residency hints. However, the PEP must understand and accept most device power management notifications even if the PEP chooses not to act on them. The PEP must refuse to handle (by returning FALSE) every notification that it does not recognize.

The *AcceptDeviceNotification* routine can be called at IRQL <= DISPATCH_LEVEL. The IRQL at which the PEP's *AcceptAcpiNotification* routine is called varies according to the type of notification that is being sent. The notification type is specified by the *Notification* parameter. For more information, see the individual reference pages under Device power management (DPM) notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKNOTIFYPPM callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptProcessorNotification* event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYPPM Pepcallbacknotifyppm;

BOOLEAN Pepcallbacknotifyppm(
  PEPHANDLE Handle,
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Handle`

The device handle for the PEP for the target processor. If **NULL**, the notification does not target a specific processor.

`Notification`

The PPM notification ID. This parameter specifies the type of PPM notification that PoFx is sending. For a list of PPM notification IDs, see Processor power management (PPM) notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various PPM notification IDs, see Processor power management (PPM) notifications.

Return Value

The *AcceptProcessorNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a processor. The **AcceptProcessorNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptProcessorNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptProcessorNotification* routine with PoFx.

The IRQL at which the PEP's *AcceptProcessorNotification* routine is called varies according to the type of notification that is being sent. The context in which this interface is called ranges from PASSIVE_LEVEL to HIGH_LEVEL with interrupts disabled. For more information about the IRQL at which each notification type is sent, see the individual reference pages under Processor power management (PPM) notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL to HIGH_LEVEL

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKPOWERONCRASHDUMPDEVICE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *PowerOnDumpDeviceCallback* callback routine turns on the crash-dump device.

Syntax

```
PEPCALLBACKPOWERONCRASHDUMPDEVICE Pepcallbackpoweroncrashdumpdevice;

BOOLEAN Pepcallbackpoweroncrashdumpdevice(
  PPEP_CRASHDUMP_INFORMATION CrashdumpInformation
)
{...}
```

Parameters

`CrashdumpInformation`

A pointer to a PEP_CRASHDUMP_INFORMATION structure that contains information about the crash-dump device.

Return Value

This routine returns TRUE if it succeeds in turning on power to the crash-dump device. Otherwise, it returns FALSE.

Remarks

This routine is implemented by the platform extension plug-in (PEP). The routine handles requests from the Windows kernel to turn on the crash-dump device so that a crash dump can be saved to disk.

The PEP registers its *PowerOnDumpDeviceCallback* routine in response to a PEP_DPM_REGISTER_CRASHDUMP_DEVICE notification from the Windows power management framework (PoFx).

The *PowerOnDumpDeviceCallback* routine is called at IRQL = HIGH_LEVEL with interrupts disabled. Thus, this routine cannot use timers, events, or any synchronization primitives other than spin locks.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

IRQL	Called at HIGH_LEVEL.

See Also

PEP_CRASHDUMP_INFORMATION

PEP_DPM_REGISTER_CRASHDUMP_DEVICE

POFXCALLBACKCRITICALRESOURCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **TransitionCriticalResource** routine changes the active/inactive state of a core system component.

Syntax

```
POFXCALLBACKCRITICALRESOURCE Pofxcallbackcriticalresource;

void Pofxcallbackcriticalresource(
  POHANDLE DeviceHandle,
  ULONG Component,
  BOOLEAN Active
)
{...}
```

Parameters

`DeviceHandle`

A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`Active`

Whether to transition the device from inactive to active, or vice versa. Set to TRUE to transition the device from inactive to active, or to FALSE to transition the device from active to inactive.

Return Value

None

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **TransitionCriticalResource** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **TransitionCriticalResource** routine.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS callback function

1/25/2019 • 2 minutes to read • Edit Online

The **EnumerateUnmaskedInterrupts** routine enumerates interrupt sources whose interrupts are unmasked and enabled.

Syntax

```
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS Pofxcallbackenumerateunmaskedinterrupts;

NTSTATUS Pofxcallbackenumerateunmaskedinterrupts(
  POHANDLE PluginHandle,
  ULONG EnumerateFlags,
  PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK Callback,
  PVOID CallbackContext,
  PPEP_UNMASKED_INTERRUPT_INFORMATION InterruptInformation
)
{...}
```

Parameters

`PluginHandle`

A POHANDLE value. If non-NULL, this parameter is a handle that identifies the platform extension plug-in (PEP), in which case **EnumerateUnmaskedInterrupts** enumerates only interrupts that are managed by this PEP. If this parameter is NULL, **EnumerateUnmaskedInterrupts** enumerates *all* interrupts in the hardware platform that are unmasked and enabled.

`EnumerateFlags`

No flags are currently defined. Set this parameter to PEP_ENUMERATE_UNMASKED_INTERRUPT_FLAGS_NONE (0x0).

`Callback`

A pointer to a caller-implemented EnumerateInterruptSource callback routine. This callback routine is called once for each interrupt source whose interrupt is unmasked. These callbacks occur synchronously before the **EnumerateUnmaskedInterrupts** routine returns.

`CallbackContext`

A pointer to a callback context. This pointer is passed as a parameter to the *EnumerateInterruptSource* callback routine pointed to by the *Callback* parameter. The contents of the callback context are PEP-defined, and are opaque to the power management framework (PoFx).

`InterruptInformation`

A pointer to a caller-allocated buffer whose size is at least **sizeof**(PEP_UNMASKED_INTERRUPT_INFORMATION) bytes. **EnumerateUnmaskedInterrupts** will use this buffer to transfer interrupt information to the PEP during calls to the PEP's *EnumerateInterruptSource* callback routine.

Return Value

EnumerateUnmaskedInterrupts returns STATUS_SUCCESS if the call successfully enumerates the interrupts. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The **Size** or **Version** member of the **PEP_UNMASKED_INTERRUPT_INFORMATION** structure contains an invalid value.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **EnumerateUnmaskedInterrupts** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **EnumerateUnmaskedInterrupts** routine.

Before the platform enters a system power state in which the interrupt controllers are power-gated, the PEP can call **EnumerateUnmaskedInterrupts** to get the information it needs to properly configure wake-up controllers for interrupts that are to remain unmasked.

The PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

EnumerateInterruptSource

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_UNMASKED_INTERRUPT_INFORMATION

POFXCALLBACKPLATFORMIDLEVETO callback function

1/25/2019 • 2 minutes to read • Edit Online

The **PlatformIdleVeto** routine increments or decrements the veto count for a veto code for a platform idle state.

Syntax

```
POFXCALLBACKPLATFORMIDLEVETO Pofxcallbackplatformidleveto;

NTSTATUS Pofxcallbackplatformidleveto(
  POHANDLE ProcessorHandle,
  ULONG PlatformState,
  ULONG VetoReason,
  BOOLEAN Increment
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with power management framework (PoFx). The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`PlatformState`

The index of the platform idle state whose veto count is being incremented or decremented. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified N platform idle states, valid platform idle state indexes range from 0 to N–1.

`VetoReason`

A PEP-defined veto code that indicates why the platform idle state cannot be entered. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. If the PEP specified M veto reasons, valid veto codes range from 1 to M.

`Increment`

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Return Value

PlatformIdleVeto returns STATUS_SUCCESS if the call successfully increments or decrements the veto count. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_I MPL EME NTE D	Indicates that the **PlatformIdleVeto** routine is not implemented for this processor.

Remarks

This routine is implemented by the PoFx and is called by the PEP. The **PlatformIdleVeto** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **PlatformIdleVeto** routine.

After the PEP handles the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification, each platform idle state starts with a veto count of zero for all veto codes. The PEP can increment the veto count of a veto code to prevent the operating system from transitioning the platform to this idle state. The PEP decrements the veto count when the veto reason that caused the count to be incremented is no longer in effect. Only after the count for all veto codes has decremented to zero is the operating system allowed to transition the platform to this idle state.

This routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

POFXCALLBACKPROCESSORHALT callback function

1/25/2019 • 3 minutes to read • Edit Online

The **ProcessorHalt** routine prepares the processor to be halted.

Syntax

```
POFXCALLBACKPROCESSORHALT Pofxcallbackprocessorhalt;

NTSTATUS Pofxcallbackprocessorhalt(
  ULONG Flags,
  PVOID Context,
  PPROCESSOR_HALT_ROUTINE Halt
)
{...}
```

Parameters

`Flags`

Flags that indicate the properties of the idle state that the processor will enter. The *Flags* parameter is set to zero or to the bitwise-OR of one or more of the following flag bits.

FLAG NAME	VALUE	DESCRIPTION
PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE	0x01	The PEP's *Halt* callback routine is responsible for flushing the processor's caches. If this flag is not set, **ProcessorHalt** will flush the caches on entry to the processor idle state, and will invalidate the caches on exit from this state.
PROCESSOR_HALT_CACHE_COHERENT	0x02	Prepare the processor to enter an idle state in which the processor's caches will remain coherent.
PROCESSOR_HALT_CONTEXT_RETAINED	0x04	Prepare the processor to enter an idle state in which the processor's thread context will be preserved.
PROCESSOR_HALT_RETURN_NOT_SAFE	0x08	The PEP's *Halt* callback routine is guaranteed to not return. Set this flag if the *Halt* callback transitions the processor to a context-losing, power-gated state that cannot be canceled by hardware. When this flag is set, the operating system will treat a return from the *Halt* callback as a fatal error.
PROCESSOR_HALT_VIA_PSCI_CPU_SUSPEND	0x16	

`Context`

A pointer to a PEP-defined processor-halt context. This pointer is passed as a parameter to the *Halt* callback routine. This context is opaque to the Windows power management framework (PoFx).

`Halt`

A pointer to a PEP-implemented *Halt* callback routine. PoFx calls this routine after preparations to halt the processor have been completed. During this callback, the PEP is expected to transition the processor to the *halted* state.

Return Value

ProcessorHalt returns STATUS_SUCCESS if the processor is successfully prepared to be halted. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The *Halt* parameter is NULL; or an invalid flag value was specified in *Flags*; or *Flags* contains an illegal combination of flag bits. For more information, see Remarks.
STA TUS _UN SUC CES SFU L	The PEP's *Halt* callback routine unexpectedly returned from an idle state in which the processor's hardware context was not preserved.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **ProcessorHalt** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **ProcessorHalt** routine.

Before halting the processor, the PEP calls the **ProcessorHalt** routine to give PoFx an opportunity to save the processor's hardware context. If necessary, **ProcessorHalt** saves this state internally in PoFx so that the state can later be restored when the processor exits the idle state. After preparing the processor to enter the idle state, **ProcessorHalt** calls the PEP's *Halt* callback routine to halt the processor.

As part of the PEP's handling of a PEP_NOTIFY_PPM_IDLE_EXECUTE notification, the PEP must transition the processor to the idle state that the PEP has selected. The following are the two ways to enter the processor idle state:

- For a processor idle state in which the processor's caches remain coherent so that all system and processor state is maintained, the PEP can enter the idle state directly without first calling **ProcessorHalt**.
- For a processor idle state in which the processor's caches might not remain coherent, or an idle state in which the processor hardware context is not preserved, the PEP must call **ProcessorHalt** before transitioning the processor to the idle state.

The following combinations of flag bits are illegal:

- PROCESSOR_HALT_CONTEXT_RETAINED = 1 and PROCESSOR_HALT_RETURN_NOT_SAFE = 1

  The PEP's *Halt* callback routine must return from any state in which context is retained.

- PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE = 1 and PROCESSOR_HALT_CACHE_COHERENT = 1

  The cache-flush-override flag should be set only if entering an idle state that is not cache coherent.

- PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE = 0 and PROCESSOR_HALT_CACHE_COHERENT = 0

  The cache-flush-override flag must be set for any non-cache coherent halt.

- PROCESSOR_HALT_CONTEXT_RETAINED = 0 and PROCESSOR_HALT_CACHE_COHERENT = 1

  Any idle states that lose processor hardware context (and therefore use the multiprocessor parking protocol to exit the idle state and return control to the operating system) are not cache-coherent states.

If the *Flags* parameter contains an illegal combination of flag bits, **ProcessorHalt** fails and returns STATUS_INVALID_PARAMETER.

The PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_IDLE_EXECUTE

POFXCALLBACKPROCESSORIDLEVETO callback function

1/25/2019 • 2 minutes to read • Edit Online

The **ProcessorIdleVeto** routine increments or decrements the veto count for a veto code for a processor idle state.

Syntax

```
POFXCALLBACKPROCESSORIDLEVETO Pofxcallbackprocessoridleveto;

NTSTATUS Pofxcallbackprocessoridleveto(
  POHANDLE ProcessorHandle,
  ULONG ProcessorState,
  ULONG VetoReason,
  BOOLEAN Increment
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with power management framework (PoFx). The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`ProcessorState`

The index of the processor idle state whose veto count is being incremented or decremented. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor idle state indexes range from 0 to N–1.

`VetoReason`

A PEP-defined veto code that indicates why the processor idle state cannot be entered. The PEP previously specified the supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. If the PEP specified M veto reasons, valid veto codes range from 1 to M.

`Increment`

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Return Value

This routine does not return a value.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **ProcessorIdleVeto** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **ProcessorIdleVeto** routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTCOMMON callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestCommon** routine is a generic request handler.

Syntax

```
POFXCALLBACKREQUESTCOMMON Pofxcallbackrequestcommon;

NTSTATUS Pofxcallbackrequestcommon(
  ULONG RequestId,
  PVOID Data
)
{...}
```

Parameters

`RequestId`

A request ID that specifies the operation being requested.

`Data`

A pointer to a data structure that contains the input data and/or result data for the request specified by the *RequestId* parameter.

Return Value

RequestCommon returns STATUS_SUCCESS if the request is successfully handled. Otherwise, it returns an appropriate error status code.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **RequestCommon** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **RequestCommon** routine.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)

IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTINTERRUPT callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestInterrupt** routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.

Syntax

```
POFXCALLBACKREQUESTINTERRUPT Pofxcallbackrequestinterrupt;

NTSTATUS Pofxcallbackrequestinterrupt(
  ULONG Gsiv
)
{...}
```

Parameters

`Gsiv`

The global system interrupt vector (GSIV) number that identifies this interrupt. The ACPI firmware assigns GSIV numbers to all primary interrupt lines. For secondary (GPIO) interrupt lines, the GSIV number is dynamically assigned by the operating system.

Return Value

RequestInterrupt returns STATUS_SUCCESS if the specified interrupt is successfully replayed. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	Not a valid GSIV number.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **RequestInterrupt** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **RequestInterrupt** routine.

Assertion of an edge-triggered interrupt signal is transient in nature and can be lost if the power to a primary interrupt controller is gated off when the interrupt occurs. In contrast, a level-triggered interrupt signal is expected

to remain asserted until it can be serviced.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTWORKER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestWorker** routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.

Syntax

```
POFXCALLBACKREQUESTWORKER Pofxcallbackrequestworker;

void Pofxcallbackrequestworker(
  POHANDLE PluginHandle
)
{...}
```

Parameters

`PluginHandle`

A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

Return Value

None.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **RequestWorker** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **RequestWorker** routine.

Each time the PEP calls this routine, PoFx responds by sending a PEP_DPM_WORK notification to the PEP. This notification might be delayed if a worker thread is not immediately available to process the work request.

A PEP can call this routine at IRQL <= DISPATCH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

PEP_DPM_WORK

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKUPDATEPLATFORMIDLESTATE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **UpdatePlatformIdleState** routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.

Syntax

```
POFXCALLBACKUPDATEPLATFORMIDLESTATE Pofxcallbackupdateplatformidlestate;

NTSTATUS Pofxcallbackupdateplatformidlestate(
  POHANDLE ProcessorHandle,
  ULONG PlatformState,
  PPEP_PLATFORM_IDLE_STATE_UPDATE Update
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`PlatformState`

An index that identifies the platform idle state whose properties are to be updated. In response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification, the PEP specified the number of supported platform idle states. If the PEP specified N platform idle states, valid platform idle state indexes range from 0 to N-1. In response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification, the PEP specified the properties of this platform idle state.

`Update`

A pointer to a PEP_PLATFORM_IDLE_STATE_UPDATE structure that contains the updated properties of the platform idle state.

Return Value

UpdatePlatformIdleState returns STATUS_SUCCESS if it successfully updates the properties of the platform idle state. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_S UPP ORT ED	The version number in the **PEP_PLATFORM_IDLE_STATE_UPDATE** structure is not a supported value.
STA TUS _NO T_I MPL EME NTE D	The **UpdatePlatformIdleState** routine is not implemented for this processor.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **UpdatePlatformIdleState** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **UpdatePlatformIdleState** routine.

The PEP must wait until after it has completed all PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notifications to call the **UpdatePlatformIdleState** routine.

The **UpdatePlatformIdleState** routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PLATFORM_IDLE_STATE_UPDATE

POFXCALLBACKUPDATEPROCESSORIDLESTATE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **UpdateProcessorIdleState** routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.

Syntax

```
POFXCALLBACKUPDATEPROCESSORIDLESTATE Pofxcallbackupdateprocessoridlestate;

NTSTATUS Pofxcallbackupdateprocessoridlestate(
  POHANDLE ProcessorHandle,
  ULONG ProcessorState,
  PPEP_PROCESSOR_IDLE_STATE_UPDATE Update
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`ProcessorState`

An index that identifies the processor idle state whose properties are to be updated. In response to a previous PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification, the PEP specified the number of supported processor idle states and the properties of these states. If the PEP specified N processor idle states, valid processor idle state indexes range from 0 to N-1.

`Update`

A pointer to a PEP_PROCESSOR_IDLE_STATE_UPDATE structure that contains the updated properties of the processor idle state.

Return Value

UpdateProcessorIdleState returns STATUS_SUCCESS if it successfully updates the properties of the processor idle state. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_S UPP ORT ED	The version number in the **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure is not a supported value.
STA TUS _NO T_I MPL EME NTE D	The **UpdateProcessorIdleState** routine is not implemented for this processor.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **UpdateProcessorIdleState** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **UpdateProcessorIdleState** routine.

The **UpdateProcessorIdleState** routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_PROCESSOR_IDLE_STATE_UPDATE

PoFxRegisterCoreDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterCoreDevice** routine registers a new core system resource with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterCoreDevice(
  PCUNICODE_STRING   Id,
  PPO_FX_CORE_DEVICE Device,
  POHANDLE           *Handle
);
```

Parameters

`Id`

A string that uniquely identifies the core system resource. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`Device`

A pointer to a PO_FX_CORE_DEVICE structure that describes the power characteristics of the core system resource.

`Handle`

A pointer to a location to which the routine writes a POHANDLE value. This handle represents the registration of the core system resource with PoFx.

Return Value

PoFxRegisterCoreDevice returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The component count for this device is zero, or the **PO_FX_CORE_DEVICE** structure contains an invalid version number, or the **ComponentCriticalTransitionCallback** member of this structure contains an invalid function pointer.

STA TUS _INS UFFI CIE NT_ RES OU RCE S	Unable to allocate the resources required to complete the requested registration.

Remarks

This routine registers a core system resource so that this resource can be power-managed by PoFx.

Core system resources are hardware devices, such as timers and interrupt controllers, that are managed by the Windows hardware abstraction layer (HAL). These devices provide basic functions that are required by the operating system. Due to the close relationship between core system resources and processors, the power management of these resources needs to be coordinated with processor idle state management.

The **PoFxRegisterCoreDevice** routine should be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PO_FX_CORE_DEVICE

UNICODE_STRING

PoFxRegisterPlugin function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterPlugin** routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterPlugin(
  PPEP_INFORMATION        PepInformation,
  PPEP_KERNEL_INFORMATION KernelInformation
);
```

Parameters

`PepInformation`

A pointer to a PEP_INFORMATION structure.

`KernelInformation`

A pointer to a PEP_KERNEL_INFORMATION_STRUCT_V3 structure.

Return Value

PoFxRegisterPlugin returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The **Version** or **Size** member of the **PEP_KERNEL_INFORMATION** structure is set to an invalid value; or the **AcceptDeviceNotification** member of this structure is set to NULL.
STA TUS _IN VAL ID_P EP_I NFO _VE RSI ON	The **Version** member of the **PEP_INFORMATION** structure is set to an invalid value.

STA TUS _INS UFFI CIE NT_ RES OU RCE S	Unable to allocate the resources required to complete the requested registration.

Remarks

A PEP calls this routine to register itself with PoFx.

A PEP cannot unregister, and cannot register twice. If the PEP must be serviced, the operating system must restart.

The PoFxRegisterPluginEx routine is similar to **PoFxRegisterPlugin**, except that it takes an additional parameter, *Flags*.

The PEP must call **PoFxRegisterPlugin** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PEP_INFORMATION

PEP_KERNEL_INFORMATION_STRUCT_V3

PoFxRegisterPluginEx

PoFxRegisterPluginEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterPluginEx** routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterPluginEx(
  PPEP_INFORMATION        PepInformation,
  ULONGLONG               Flags,
  PPEP_KERNEL_INFORMATION KernelInformation
);
```

Parameters

`PepInformation`

A pointer to a PEP_INFORMATION structure that contains pointers to one or more callback routines that are implemented by the PEP. These routines handle notifications that are sent to the PEP by PoFx.

`Flags`

A set of flag bits for configuring the PEP interface. Set this member to zero or to the following value.

FLAG BIT	DESCRIPTION
PEP_FLAG_WORKER_CONCURRENCY	

`KernelInformation`

A pointer to a PEP_KERNEL_INFORMATION structure.

Return Value

PoFxRegisterPluginEx returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The **Version** or **Size** member of the **PEP_KERNEL_INFORMATION** structure is set to an invalid value; or the **AcceptDeviceNotification** member of this structure is set to NULL.

STA TUS _IN VAL ID_P EP_I NFO _VE RSI ON	The **Version** member of the **PEP_INFORMATION** structure is set to an invalid value.
STA TUS _INS UFFI CIE NT_ RES OU RCE S	Unable to allocate the resources required to complete the requested registration.

Remarks

A PEP calls this routine to register itself with PoFx.

A PEP cannot unregister, and cannot register twice. If the PEP must be serviced, the operating system must restart.

The PoFxRegisterPlugin routine is similar to **PoFxRegisterPluginEx**, except that it does not take a *Flags* parameter.

The PEP must call **PoFxRegisterPluginEx** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PEP_INFORMATION

PEP_KERNEL_INFORMATION

PoFxRegisterPlugin

PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

An *EnumerateInterruptSource* callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.

Syntax

```
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK PpoEnumerateInterruptSourceCallback;

BOOLEAN PpoEnumerateInterruptSourceCallback(
  PVOID CallbackContext,
  PPEP_UNMASKED_INTERRUPT_INFORMATION InterruptInformation
)
{...}
```

Parameters

`CallbackContext`

A pointer to a callback context. The PEP specified this pointer value as a parameter in the EnumerateUnmaskedInterrupts call that initiated the enumeration of interrupt sources.

`InterruptInformation`

A pointer to a PEP_UNMASKED_INTERRUPT_INFORMATION structure that contains information about the interrupt source.

Return Value

If the *EnumerateInterruptSource* callback routine returns TRUE, **EnumerateUnmaskedInterrupts** will continue to call the *EnumerateInterruptSource* callback routine while more interrupts are available to be enumerated. If the *EnumerateInterruptSource* callback routine returns FALSE, **EnumerateUnmaskedInterrupts** returns without enumerating any more interrupts.

Remarks

This callback routine is implemented by a PEP, and is called by the Windows power management framework (PoFx). The *Callback* parameter of the EnumerateUnmaskedInterrupts routine is a pointer to an *EnumerateInterruptSource* callback routine.

After a PEP calls **EnumerateUnmaskedInterrupts** to enumerate the unmasked interrupt sources, PoFx calls the PEP's *EnumerateInterruptSource* callback routine once for each unmasked interrupt source. **EnumerateUnmaskedInterrupts** returns only after the last call to the *EnumerateInterruptSource* callback routine completes.

An *EnumerateInterruptSource* callback routine is called at the same IRQL as the PEP's call to **EnumerateUnmaskedInterrupts** that initiates the enumeration callbacks.

Requirements

Target Platform	Windows
Header	pep_x.h (include Pep_x.h)
IRQL	See Remarks.

See Also

PEP_UNMASKED_INTERRUPT_INFORMATION

EnumerateUnmaskedInterrupts

pepevents.h header

1/8/2019 • 2 minutes to read • Edit Online

This is an include file generated by Message Compiler. For information about the structures and macros, see the comments in the header.

- Windows kernel

pepfx.h header

1/8/2019 • 13 minutes to read • Edit Online

This header contains programming interfaces used by the Power Engine Plugins (PEPs). For more information, see:

- Platform Extension Plug-ins (PEPs) pepfx.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.
PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE	The PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.
PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE	The PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.
PEP_ACPI_INITIALIZE_IOPORT_RESOURCE	The PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_MEMORY_RESOURCE	The PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.
PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE	The PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

TITLE	DESCRIPTION
PoFxRegisterCoreDevice	The PoFxRegisterCoreDevice routine registers a new core system resource with the Windows power management framework (PoFx).
PoFxRegisterPlugin	The PoFxRegisterPlugin routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).
PoFxRegisterPluginEx	The PoFxRegisterPluginEx routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Callback functions

TITLE	DESCRIPTION
PEPCALLBACKNOTIFYACPI	An AcceptAcpiNotification event callback routine handles ACPI notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYDPM	An AcceptDeviceNotification event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKNOTIFYPPM	An AcceptProcessorNotification event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).
PEPCALLBACKPOWERONCRASHDUMPDEVICE	The PowerOnDumpDeviceCallback callback routine turns on the crash-dump device.
POFXCALLBACKCRITICALRESOURCE	The TransitionCriticalResource routine changes the active/inactive state of a core system component.
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS	The EnumerateUnmaskedInterrupts routine enumerates interrupt sources whose interrupts are unmasked and enabled.
POFXCALLBACKPLATFORMIDLEVETO	The PlatformIdleVeto routine increments or decrements the veto count for a veto code for a platform idle state.
POFXCALLBACKPROCESSORHALT	The ProcessorHalt routine prepares the processor to be halted.
POFXCALLBACKPROCESSORIDLEVETO	The ProcessorIdleVeto routine increments or decrements the veto count for a veto code for a processor idle state.
POFXCALLBACKREQUESTCOMMON	The RequestCommon routine is a generic request handler.
POFXCALLBACKREQUESTINTERRUPT	The RequestInterrupt routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.

TITLE	DESCRIPTION
POFXCALLBACKREQUESTWORKER	The RequestWorker routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.
POFXCALLBACKUPDATEPLATFORMIDLESTATE	The UpdatePlatformIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.
POFXCALLBACKUPDATEPROCESSORIDLESTATE	The UpdateProcessorIdleState routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK	An EnumerateInterruptSource callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.

Structures

TITLE	DESCRIPTION
PEP_ABANDON_DEVICE	The PEP_ABANDON_DEVICE structure identifies a device that has been abandoned and will no longer be used by the operating system.
PEP_ACPI_ABANDON_DEVICE	The PEP_ACPI_ABANDON_DEVICE structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.
PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE	The PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure contains an enumeration of the objects in the namespace of the device.
PEP_ACPI_EVALUATE_CONTROL_METHOD	The PEP_ACPI_EVALUATE_CONTROL_METHOD structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.
PEP_ACPI_EXTENDED_ADDRESS	The PEP_ACPI_EXTENDED_ADDRESS structure is used to report resource usage in the address space such as memory and IO.
PEP_ACPI_GPIO_RESOURCE	The PEP_ACPI_GPIO_RESOURCE structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.
PEP_ACPI_INTERRUPT_RESOURCE	The PEP_ACPI_INTERRUPT_RESOURCE structure describes an ACPI interrupt resource.
PEP_ACPI_IO_MEMORY_RESOURCE	The PEP_ACPI_IO_MEMORY_RESOURCE structure describes an ACPI IO port descriptor resource.
PEP_ACPI_OBJECT_NAME	The PEP_ACPI_OBJECT_NAME union contains the four-character name of an ACPI object.

TITLE	DESCRIPTION
PEP_ACPI_OBJECT_NAME_WITH_TYPE	The PEP_ACPI_OBJECT_NAME_WITH_TYPE structure that specifies both the path-relative name of an ACPI object and the type of this object.
PEP_ACPI_PREPARE_DEVICE	The PEP_ACPI_PREPARE_DEVICE structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.
PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure contains a list of raw resources that are needed to control power to the device.
PEP_ACPI_QUERY_OBJECT_INFORMATION	The PEP_ACPI_QUERY_OBJECT_INFORMATION structure contains information about an ACPI object.
PEP_ACPI_REGISTER_DEVICE	The PEP_ACPI_REGISTER_DEVICE structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.
PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES	The PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.
PEP_ACPI_RESOURCE	The PEP_ACPI_RESOURCE structure contains hardware details for a specific ACPI resource.
PEP_ACPI_RESOURCE_FLAGS	The PEP_ACPI_RESOURCE_FLAGS structure contains flags describing an ACPI resource.
PEP_ACPI_SPB_I2C_RESOURCE	The PEP_ACPI_SPB_I2C_RESOURCE structure describes an ACPI I2C serial bus resource.
PEP_ACPI_SPB_RESOURCE	The PEP_ACPI_SPB_RESOURCE structure describes an ACPI serial bus connection resource.
PEP_ACPI_SPB_SPI_RESOURCE	The PEP_ACPI_SPB_SPI_RESOURCE structure describes an ACPI SPI serial bus resource.
PEP_ACPI_SPB_UART_RESOURCE	The PEP_ACPI_SPB_UART_RESOURCE structure describes an ACPI UART serial bus resource.
PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES	The PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.
PEP_ACPI_UNREGISTER_DEVICE	The PEP_ACPI_UNREGISTER_DEVICE structure contains information about a device that has been unregistered from ACPI services.
PEP_COMPONENT_PERF_INFO	The PEP_COMPONENT_PERF_INFO structure describes the performance states (P-states) of a component.
PEP_COMPONENT_PERF_SET	The PEP_COMPONENT_PERF_SET structure describes the performance states (P-states) in a P-state set.

TITLE	DESCRIPTION
PEP_COMPONENT_PERF_STATE_REQUEST	The PEP_COMPONENT_PERF_STATE_REQUEST structure specifies a performance state (P-state) set and a new performance level to assign to this set.
PEP_COMPONENT_PLATFORM_CONSTRAINTS	The PEP_COMPONENT_PLATFORM_CONSTRAINTS structure describes the lowest-powered Fx state of that a component can be in when the platform is in a particular idle state.
PEP_COMPONENT_V2	The PEP_COMPONENT_V2 structure specifies the power state attributes of a component in the device.
PEP_COORDINATED_DEPENDENCY_OPTION	The PEP_COORIDNATED_DEPENDENCY_OPTION structure describes a coordinated idle state's dependency to the OS.
PEP_COORDINATED_IDLE_STATE	The PEP_COORIDNATED_IDLE_STATE structure describes a coordinated idle state to the OS.
PEP_CRASHDUMP_INFORMATION	The PEP_CRASHDUMP_INFORMATION structure contains information about a crash-dump device.
PEP_DEBUGGER_TRANSITION_REQUIREMENTS	The PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure indicates the platform idle states for which the debugger device must be turned on.
PEP_DEVICE_PLATFORM_CONSTRAINTS	The PEP_DEVICE_PLATFORM_CONSTRAINTS structure specifies the constraints for entry to the various Dx power states that are supported by a device.
PEP_DEVICE_POWER_STATE	The PEP_DEVICE_POWER_STATE structure indicates the status of a transition to a new Dx (device power) state.
PEP_DEVICE_REGISTER_V2	The PEP_DEVICE_REGISTER structure describes all the components in a particular device.
PEP_DEVICE_STARTED	The PEP_DEVICE_STARTED structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).
PEP_INFORMATION	The PEP_INFORMATION structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).
PEP_KERNEL_INFORMATION_STRUCT_V3	The PEP_KERNEL_INFORMATION_STRUCT_V3 structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).
PEP_LOW_POWER_EPOCH	The PEP_LOW_POWER_EPOCH structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).
PEP_NOTIFY_COMPONENT_IDLE_STATE	The PEP_NOTIFY_COMPONENT_IDLE_STATE structure contains status information about a component's pending transition to a new Fx power state.

TITLE	DESCRIPTION
PEP_PERF_STATE	The PEP_PERF_STATE structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.
PEP_PLATFORM_IDLE_STATE	The PEP_PLATFORM_IDLE_STATE structure specifies the properties of a platform idle state.
PEP_PLATFORM_IDLE_STATE_UPDATE	The PEP_PLATFORM_IDLE_STATE_UPDATE structure contains the updated properties of a platform idle state.
PEP_POWER_CONTROL_COMPLETE	The PEP_POWER_CONTROL_COMPLETE structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.
PEP_POWER_CONTROL_REQUEST	The PEP_POWER_CONTROL_REQUEST structure contains a request from a driver for a power control operation.
PEP_PPM_CONTEXT_QUERY_PARKING_PAGE	The PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure describes the parking page for a processor.
PEP_PPM_CST_STATE	The PEP_PPM_CST_STATE structure specifies the properties of a C state (ACPI processor power state).
PEP_PPM_CST_STATES	The PEP_PPM_CST_STATES structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.
PEP_PPM_ENTER_SYSTEM_STATE	Used in the PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE notification to notify PEP that the system is about to enter a system power state. .
PEP_PPM_FEEDBACK_READ	The PEP_PPM_FEEDBACK_READ structure contains the value read from a processor performance feedback counter.
PEP_PPM_IDLE_COMPLETE	The PEP_PPM_IDLE_COMPLETE structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_COMPLETE_V2	The PEP_PPM_IDLE_COMPLETE_V2 structure describe the idle states from which the processor and hardware platform are waking.
PEP_PPM_IDLE_EXECUTE	The PEP_PPM_IDLE_EXECUTE structure specifies the idle state that the processor is to enter.
PEP_PPM_IDLE_EXECUTE_V2	The PEP_PPM_IDLE_EXECUTE_V2 structure specifies the idle state that the processor is to enter.
PEP_PPM_INITIATE_WAKE	The PEP_PPM_INITIATE_WAKE structure indicates whether a processor requires an interrupt to wake up from an idle state.
PEP_PPM_IS_PROCESSOR_HALTED	The PEP_PPM_IS_PROCESSOR_HALTED structure indicates whether the processor is currently halted in its selected idle state.

TITLE	DESCRIPTION
PEP_PPM_PARK_MASK	The PEP_PROCESSOR_PARK_MASK structure contains the current core parking mask.
PEP_PPM_PARK_SELECTION	The PEP_PPM_PARK_SELECTION structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PARK_SELECTION_V2	The PEP_PPM_PARK_SELECTION_V2 structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.
PEP_PPM_PERF_CHECK_COMPLETE	The PEP_PPM_PERF_CHECK_COMPLETE structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.
PEP_PPM_PERF_CONSTRAINTS	The PEP_PPM_PERF_CONSTRAINTS structure describes the performance limits to apply to the processor.
PEP_PPM_PERF_SET	The PEP_PPM_PERF_SET structure specifies the new performance level that the operating system is requesting for the processor.
PEP_PPM_PERF_SET_STATE	Used in the PEP_NOTIFY_PPM_PERF_SET notification at runtime to set the current operating performance of the processor. .
PEP_PPM_PLATFORM_STATE_RESIDENCIES	The PEP_PPM_PLATFORM_STATE_RESIDENCIES structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.
PEP_PPM_PLATFORM_STATE_RESIDENCY	The PEP_PPM_PLATFORM_STATE_RESIDENCY structure specifies the accumulated residency time and transition count for a particular platform idle state.
PEP_PPM_QUERY_CAPABILITIES	The PEP_PPM_QUERY_CAPABILITIES structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).
PEP_PPM_QUERY_COORDINATED_DEPENDENCY	The PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure describes dependencies for coordinated idle states.
PEP_PPM_QUERY_COORDINATED_STATES	The PEP_PPM_QUERY_COORDINATED_STATES structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.

TITLE	DESCRIPTION
PEP_PPM_QUERY_DISCRETE_PERF_STATES	Used in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification that stores the list of discrete performance states that PEP supports, if the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification indicates support for discrete performance states. .
PEP_PPM_QUERY_DOMAIN_INFO	Used in the PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO notification that queries for information about a performance domain. .
PEP_PPM_QUERY_FEEDBACK_COUNTERS	The PEP_PPM_QUERY_FEEDBACK_COUNTERS structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.
PEP_PPM_QUERY_IDLE_STATES_V2	The PEP_PPM_QUERY_IDLE_STATES_V2 structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.
PEP_PPM_QUERY_PERF_CAPABILITIES	The PEP_PPM_QUERY_PERF_CAPABILITIES structure describes the performance capabilities of the processors in the specified processor performance domain.
PEP_PPM_QUERY_PLATFORM_STATE	The PEP_PPM_QUERY_PLATFORM_STATE structure contains information about a platform idle state.
PEP_PPM_QUERY_PLATFORM_STATES	The PEP_PPM_QUERY_PLATFORM_STATES structure specifies the number of platform idle states the hardware platform supports.
PEP_PPM_QUERY_STATE_NAME	The PEP_PPM_QUERY_STATE_NAME structure contains information about a specific coordinated or platform idle state.
PEP_PPM_QUERY_VETO_REASON	The PEP_PPM_QUERY_VETO_REASON structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.
PEP_PPM_QUERY_VETO_REASONS	The PEP_PPM_QUERY_VETO_REASONS structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.
PEP_PPM_RESUME_FROM_SYSTEM_STATE	Used by the PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE notification that notifies the PEP that the system has just resumed from a system power state.
PEP_PPM_TEST_IDLE_STATE	The PEP_PPM_TEST_IDLE_STATE structure contains information about whether the processor can immediately enter a processor idle state.
PEP_PREPARE_DEVICE	The PEP_PREPARE_DEVICE structure identifies a device that must be started up in preparation for its use by the operating system.

TITLE	DESCRIPTION
PEP_PROCESSOR_FEEDBACK_COUNTER	The PEP_PROCESSOR_FEEDBACK_COUNTER structure describes a feedback counter to the operating system.
PEP_PROCESSOR_IDLE_DEPENDENCY	The PEP_PROCESSOR_IDLE_DEPENDENCY structure specifies the dependencies of a platform idle state on the specified processor.
PEP_PROCESSOR_IDLE_STATE_UPDATE	The PEP_PROCESSOR_IDLE_STATE_UPDATE structure contains the updated properties of a processor idle state.
PEP_PROCESSOR_IDLE_STATE_V2	The PEP_PROCESSOR_IDLE_STATE_V2 structure describes a processor idle state that the platform extension plug-in (PEP) supports.
PEP_PROCESSOR_PARK_PREFERENCE	The PEP_PROCESSOR_PARK_PREFERENCE structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.
PEP_PROCESSOR_PARK_STATE	The PEP_PROCESSOR_PARK_STATE structure describes the parking state for a single processor.
PEP_PROCESSOR_PERF_STATE	Use in the PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES notification. This structure describes the properties of a single performance state. .
PEP_QUERY_COMPONENT_PERF_CAPABILITIES	The PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure specifies the number of performance state (P-state) sets that are defined for a component.
PEP_QUERY_COMPONENT_PERF_SET	The PEP_QUERY_COMPONENT_PERF_SET structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_SET_NAME	The PEP_QUERY_COMPONENT_PERF_SET_NAME structure contains query information about a set of performance state values (P-state set) for a component.
PEP_QUERY_COMPONENT_PERF_STATES	The PEP_QUERY_COMPONENT_PERF_STATES structure contains a list of discrete performance state (P-state) values for the specified P-state set.
PEP_QUERY_CURRENT_COMPONENT_PERF_STATE	The PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure contains information about the current P-state in the specified P-state set.
PEP_QUERY_SOC_SUBSYSTEM	The PEP_QUERY_SOC_SUBSYSTEM structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.

TITLE	DESCRIPTION
PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME	The PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.
PEP_QUERY_SOC_SUBSYSTEM_COUNT	The PEP_QUERY_SOC_SUBSYSTEM_COUNT structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.
PEP_QUERY_SOC_SUBSYSTEM_METADATA	The PEP_QUERY_SOC_SUBSYSTEM_METADATA structure is used with the PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.
PEP_REGISTER_COMPONENT_PERF_STATES	The PEP_REGISTER_COMPONENT_PERF_STATES structure describes the performance states (P-states) of the specified component.
PEP_REGISTER_CRASHDUMP_DEVICE	The PEP_REGISTER_CRASHDUMP_DEVICE structure provides a callback routine to turn on a crash-dump device.
PEP_REGISTER_DEBUGGER	The PEP_REGISTER_DEBUGGER structure identifies a registered device that is a core system resource that provides debugger transport.
PEP_REGISTER_DEVICE_V2	The PEP_REGISTER_DEVICE_V2 structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).
PEP_REQUEST_COMPONENT_PERF_STATE	The PEP_REQUEST_COMPONENT_PERF_STATE structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).
PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING	The PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.
PEP_SOC_SUBSYSTEM_METADATA	The PEP_SOC_SUBSYSTEM_METADATA structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).
PEP_SYSTEM_LATENCY	The PEP_SYSTEM_LATENCY structure specifies the new value for the system latency tolerance.
PEP_UNMASKED_INTERRUPT_FLAGS	The PEP_UNMASKED_INTERRUPT_FLAGS union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.

TITLE	DESCRIPTION
PEP_UNMASKED_INTERRUPT_INFORMATION	The PEP_UNMASKED_INTERRUPT_INFORMATION structure contains information about an interrupt source.
PEP_UNREGISTER_DEVICE	The PEP_UNREGISTER_DEVICE structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).
PEP_WORK	The PEP_WORK structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).
PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE	The PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).
PEP_WORK_ACPI_NOTIFY	The PEP_WORK_ACPI_NOTIFY structure contains the ACPI Notify code for a device that has generated a hardware event.
PEP_WORK_COMPLETE_IDLE_STATE	The PEP_WORK_COMPLETE_IDLE_STATE structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new Fx power state.
PEP_WORK_COMPLETE_PERF_STATE	The PEP_WORK_COMPLETE_PERF_STATE structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.
PEP_WORK_INFORMATION	The PEP_WORK_INFORMATION structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).
PEP_WORK_POWER_CONTROL	The PEP_WORK_POWER_CONTROL structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.
PO_FX_CORE_DEVICE	The PO_FX_CORE_DEVICE structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.

Enumerations

TITLE	DESCRIPTION
GPIO_PIN_CONFIG_TYPE	The GPIO_PIN_CONFIG_TYPE enumeration describes a connection IO resource.
GPIO_PIN_IORESTRICTION_TYPE	The GPIO_PIN_IORESTRICTION_TYPE enumeration describes the functions that a GPIO pin is limited to performing.
PEP_ACPI_OBJECT_TYPE	The PEP_ACPI_OBJECT_TYPE enumeration indicates the type of ACPI object.

TITLE	DESCRIPTION
PEP_ACPI_RESOURCE_TYPE	The PEP_ACPI_RESOURCE_TYPE enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.
PEP_DEVICE_ACCEPTANCE_TYPE	The PEP_DEVICE_ACCEPTANCE_TYPE enumeration indicates whether a PEP accepts ownership of a device.
PEP_PERF_STATE_TYPE	The PEP_PERF_STATE_TYPE enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.
PEP_PERF_STATE_UNIT	The PEP_PERF_STATE_UNIT enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.
PEP_WORK_TYPE	The PEP_WORK_TYPE enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.

GPIO_PIN_CONFIG_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **GPIO_PIN_CONFIG_TYPE** enumeration describes a connection IO resource.

Syntax

```
typedef enum _GPIO_PIN_CONFIG_TYPE {
  PullDefault,
  PullUp,
  PullDown,
  PullNone
} GPIO_PIN_CONFIG_TYPE;
```

Constants

PullDefault	Indicates that no configuration is applied to this pin.
PullUp	Indicates that this pin is configured to use a pull-up resistor.
PullDown	Indicates that this pin is configured to use a pull-down resistor.
PullNone	Indicates that this pin is not configured to use a pull-up or pull-down resistor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

GPIO_PIN_IORESTRICTION_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **GPIO_PIN_IORESTRICTION_TYPE** enumeration describes the functions that a GPIO pin is limited to performing.

Syntax

```
typedef enum _GPIO_PIN_IORESTRICTION_TYPE {
  IoRestrictionNone,
  IoRestrictionInputOnly,
  IoRestrictionOutputOnly,
  IoRestrictionNoneAndPreserve
} GPIO_PIN_IORESTRICTION_TYPE;
```

Constants

IoRestrictionNone	Indicates that the GPIO pin is not restricted to either input or output. When no IO restriction is described, it is assumed to be **IoRestrictionNone**.
IoRestrictionInputOnly	Indicates that the GPIO pin is restricted to input.
IoRestrictionOutputOnly	Indicates that the GPIO pin is restricted to output.
IoRestrictionNoneAndPreserve	Indicates that the GPIO pin is not restricted to either input or output and that the mode should be preserved when the driver is unloaded.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

PEP_ABANDON_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ABANDON_DEVICE** structure identifies a device that has been abandoned and will no longer be used by the operating system.

Syntax

```
typedef struct _PEP_ABANDON_DEVICE {
  PCUNICODE_STRING DeviceId;
  BOOLEAN          DeviceAccepted;
} PEP_ABANDON_DEVICE, *PPEP_ABANDON_DEVICE;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device. The PEP sets this member to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device.

Remarks

This structure is used by the PEP_DPM_ABANDON_DEVICE notification. The **DeviceId** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_ABANDON_DEVICE

UNICODE_STRING

PEP_ACPI_ABANDON_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_ABANDON_DEVICE** structure indicates whether the platform extension plug-in (PEP) accepts ownership of an abandoned device.

Syntax

```
typedef struct _PEP_ACPI_ABANDON_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  BOOLEAN          DeviceAccepted;
} PEP_ACPI_ABANDON_DEVICE, *PPEP_ACPI_ABANDON_DEVICE;
```

Members

`AcpiDeviceName`

[in] A pointer to an ANSI_STRING structure that contains the fully qualified BIOS name for the device. This name specifies the the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device specified by the **AcpiDeviceName** member. Set to TRUE if the PEP claims ownership, and to FALSE if the PEP does not own the device.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_ABANDON_DEVICE notification. This notification provides an opportunity for the PEP to clean up any remaining device state after the operating system has abandoned the device. The **AcpiDeviceName** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_NOTIFY_ACPI_ABANDON_DEVICE

PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure contains an enumeration of the objects in the namespace of the device.

Syntax

```
typedef struct _PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE {
  PEPHANDLE                       DeviceHandle;
  ULONG                           RequestFlags;
  NTSTATUS                        Status;
  ULONG                           ObjectCount;
  SIZE_T                          ObjectBufferSize;
  PEP_ACPI_OBJECT_NAME_WITH_TYPE Objects[ANYSIZE_ARRAY];
} PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE, *PPEP_ACPI_ENUMERATE_DEVICE_NAMESPACE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_EDN_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of the requested device-namespace enumeration. Set this member to STATUS_SUCCESS if the PEP succeeds in enumerating the objects in the device namespace. Set to STATUS_BUFFER_TOO_SMALL if the input value in the **TotalBufferSize** member indicates that the allocated buffer is not large enough to contain the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure and **Objects** array.

`ObjectCount`

[out] The number of elements in the **Objects** array.

`ObjectBufferSize`

`Objects`

[out] The first element in an array of PEP_ACPI_OBJECT_NAME_WITH_TYPE structures that describe the objects in the ACPI namespace of the device. If this array contains more than one element, the additional array elements follow the end of the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification. The

RequestFlags and **TotalBufferSize** members contain input values supplied by PoFx when the notification is sent. The PEP may overwrite the **TotalBufferSize** input value with an output value if the input value is less than the required output buffer size. The **Status**, **ObjectCount**, and **Objects** members contain output values that the PEP writes to the structure.

If N is the number of objects listed in the ACPI namespace under this device, the PEP should verify that the storage allocated by PoFx for the **Objects** array is large enough to contain N array elements. If N > 1, the additional N−1 array elements follow the end of the **PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE** structure. The buffer size required to contain the structure and additional array elements is calculated as follows:

sizeof PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE sizeof PEP_ACPI_OBJECT_NAME_WITH_TYPE If the input value specified in the **TotalBufferSize** member is less than the required size, the PEP overwrites the **TotalBufferSize** input value with the required size, and sets the **Status** member to STATUS_BUFFER_TOO_SMALL. In response, PoFx will allocate a buffer of the required size and send a second PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification to the PEP for this device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME_WITH_TYPE

PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_EVALUATE_CONTROL_METHOD structure

2/5/2019 • 3 minutes to read • Edit Online

The **PEP_ACPI_EVALUATE_CONTROL_METHOD** structure specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result of the evaluation.

Syntax

```
typedef struct _PEP_ACPI_EVALUATE_CONTROL_METHOD {
  PEPHANDLE              DeviceHandle;
  ULONG                  RequestFlags;
  union {
    ULONG       MethodName;
    ANSI_STRING MethodNameString;
  };
  NTSTATUS               MethodStatus;
  PVOID                  CompletionContext;
  ULONG                  InputArgumentCount;
  SIZE_T                 InputArgumentSize;
  PACPI_METHOD_ARGUMENT InputArguments;
  ULONG                  OutputArgumentCount;
  SIZE_T                 OutputArgumentSize;
  PACPI_METHOD_ARGUMENT OutputArguments;
} PEP_ACPI_EVALUATE_CONTROL_METHOD, *PPEP_ACPI_EVALUATE_CONTROL_METHOD;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of flags. This member contains one of the following values.

FLAG NAME	VALUE	MEANING
PEP_ACPI_ECM_FLAG_NONE	0x0	Not a valid control method name.
PEP_ACPI_ECM_FLAG_RELATIVE_NAME	0x1	The **MethodNameUlong** member contains a path-relative control method name.
PEP_ACPI_ECM_FLAG_FULLY_QUALIFIED_NAME	0x2	The **MethodNameString** member contains a fully qualified control method name.

`MethodName`

`MethodNameString`

`MethodStatus`

[out] An NTSTATUS value that indicates the status of the evaluation of the ACPI control method. Set this member

to STATUS_SUCCESS to indicate that the platform extension plug-in (PEP) successfully completed its evaluation of the method. Set to STATUS_NOT_SUPPORTED to indicate that the PEP does not recognize the control method. Set to STATUS_BUFFER_TOO_SMALL to indicate that the output buffer size specified by the **InputArgumentSize** member is not large enough to contain the method results.

If the PEP is to evaluate the method asychronously, set this member to STATUS_PENDING and return from the AcceptAcpiNotification callback. Later, when the evaluation is completed, the PEP calls the CompleteWork routine to notify the Windows power management framework (PoFx) that the evaluation of the control method is complete.

`CompletionContext`

[in] A pointer to a completion context value. The PEP uses this member only if the control method is evaluated asynchronously. In this case, the PEP supplies this completion context pointer in the call to the CompleteWork routine that notifies the Windows power management framework (PoFx) that the evaluation of the control method is complete. Included in the input parameters to this call is a pointer to a PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure to which the PEP has written the completion context pointer. The context is opaque to the PEP and contains data used internally by PoFx.

`InputArgumentCount`

[in] The number of input arguments in the input buffer pointed to by the **InputArguments** member. Each argument starts with an ACPI_METHOD_ARGUMENT structure that specifies the size of the argument. Currently, the maximum **InputArgumentCount** value is one.

`InputArgumentSize`

[in] The size, in bytes, of the input buffer pointed to by the **InputArguments** member.

`InputArguments`

[in] A pointer to an input buffer that contains an ACPI_METHOD_ARGUMENT structure that specifies the input argument for the specified ACPI control method.

`OutputArgumentCount`

[in] The number of output arguments in the output buffer pointed to by the **OutputArguments** member. Each argument starts with an ACPI_METHOD_ARGUMENT structure that specifies the size of the argument. Currently, the maximum allowed **OutputArgumentCount** value is one.

`OutputArgumentSize`

[in, out] On input, the size, in bytes, of the output buffer pointed to by the **OutputArguments** member. If this buffer is not large enough to contain the result of evaluating the method, the PEP overwrites the **OutputArgumentSize** input value with the required size, and sets the **MethodStatus** member to STATUS_BUFFER_TOO_SMALL.

`OutputArguments`

[in] A pointer to an output buffer to which the PEP writes an ACPI_METHOD_ARGUMENT structure that contains the result of evaluating the specified ACPI control method.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification. The **MethodStatus** member contains an output value that the PEP writes to the structure in response to this notification. The **OutputArgumentSize** member contains an input value supplied by PoFx when the notification is sent. The PEP may overwrite this input value with an output value if the input value is less than the required output buffer size. All other members of this structure contain input values that are supplied by PoFx when the notification

is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

ANSI_STRING

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_EXTENDED_ADDRESS structure

1/30/2019 • 5 minutes to read • Edit Online

The **PEP_ACPI_EXTENDED_ADDRESS** structure is used to report resource usage in the address space such as memory and IO.

Syntax

```
typedef struct _PEP_ACPI_EXTENDED_ADDRESS {
  PEP_ACPI_RESOURCE_TYPE  Type;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  UCHAR                   ResourceFlags;
  UCHAR                   GeneralFlags;
  UCHAR                   TypeSpecificFlags;
  UCHAR                   RevisionId;
  UCHAR                   Reserved;
  ULONGLONG               Granularity;
  ULONGLONG               MinimumAddress;
  ULONGLONG               MaximumAddress;
  ULONGLONG               TranslationAddress;
  ULONGLONG               AddressLength;
  ULONGLONG               TypeAttribute;
  PUNICODE_STRING         DescriptorName;
} PEP_ACPI_EXTENDED_ADDRESS, *PPEP_ACPI_EXTENDED_ADDRESS;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure describing this resource.

`ResourceFlags`

Indicates the type of resource this structure describes.

VALUE	MEANING
0	Indicates that this resource is a memory range.
1	Indicates that this resource is an IO range.
2	Indicates that this resource is a bus number range.
3-191	These values are reserved for future use.

192 - 255	These values are reserved for use by the hardware vendor.

`GeneralFlags`

A value containing the bit flags that are common to all resource types.

BIT(S)	MEANING
0 - Consumer/ Producer flag	When set, this indicates that the device consumes this resource.
1 - Decode type	When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only. When not set, indicates that this bridge positively decodes this address.
2 - Minimum address fixed	When set, indicates that the minimum address is fixed.
3 - Max address fixed	When set, indicates that the maximum address is fixed.
4 to 7 - Reserved	These bits are reserved and must be set to zero.

`TypeSpecificFlags`

The value of this member is dependent on the value in **ResourceFlags** member. The flags for each resource type are described in the tables below.

Memory Resource (**ResourceFlags** = **0**)

BIT(S)	MEANING
0 - Write status	When set, indicates that this memory range is available for reading and writing. Otherwise, this indicates that this memory range is read-only.
1 to 2 - Memory attributes (MEM)	**0** - Indicates the memory is non-cacheable. **1** - Indicates the memory is cacheable. **2** - Indicates the memory is cacheable and supports write combining. **3** - The memory is cacheable and prefetchable.
3 to 4 - Memory attributes (MTP)	These bits are only defined if this memory resource describes system RAM. **0** - Address range memory: This range is available RAM usable by the operating system. **1** - Address range reserved: This range of addresses is in use or reserved by the system and is not to be included in the allocatable memory pool of the operating system's memory manager. **2** - Address range ACPI: ACPI Reclaim Memory. This range is available RAM usable by the OS after it reads the ACPI tables. **3** - Address Range NVS: ACPI NVS Memory. This range of addresses is in use or reserved by the system and must not be used by the operating system. This range is required to be saved and restored across an NVS sleep.
5 - Memory to IO translation	**0** - Type-static: This resource is memory on the primary and secondary sides of the bridge. **1** - Type-translation: This resource is memory on the secondary side of the bridge and IO on the primary side of the bridge.
6 to 7 - Reserved	These bits are reserved and must be set to zero.

IO Resource (**ResourceFlags** = **1**)

BIT(S)	MEANING

0 to 1 - Range	**0** - Reserved. **1** - Non-ISA ranges only. This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the non-ISA IO addresses that fall within the specified window. The non-ISA IO ranges are: n100-n3FF, n500-n7FF, n900-nBFF, nD00-nFFF. This bit can only be set for bridges entirely configured through the ACPI namespace. **2** - ISA ranges only. This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the ISA IO addresses that fall within the specified window. The ISA IO ranges are: n000-n0FF, n400-n4FF, n800-n8FF, nC00-nCFF. This bit can only be set for bridges entirely configured through the ACPI namespace. **3** - The memory window covers the entire range
2 to 3 - Reserved	These bits are reserved and must be set to zero.
4 - IO to memory translation	**0** - Type-static: This resource is IO on the primary and secondary sides of the bridge. **1** - Type-translation: This resource is IO on the secondary side of the bridge and memory on the primary side of the bridge.
5 - Sparse translation	This bit is only meaningful if bit 4 (IO to memory translation) is set. **0** - Dense translation: The primary-side memory address of any specific IO port within the secondary-side range can be found using the following function. *address = port + TranslationAddress* **1** - Sparse translation: The primary-side memory address of any specific IO port within the secondary-side range can be found using the following function. *address = (((port & 0xFFFc) << 10) || (port & 0xFFF)) + TranslationAddress* In the address used to access the IO port, bits 2 to 11 must be identical to bits 12 to 21, this gives four bytes of IO ports on each 4 KB page.
6 to 7 - Reserved	These bits are reserved and must be set to zero.

Bus Number Range Resource (**ResourceFlags** = **2**)

BIT(S)	MEANING

0 to 7 - Reserved	These bits are reserved and must be set to zero.

`RevisionId`

Indicates the revision of the extended address space descriptor detailed by this structure. For ACPI 3.0, this value is 1.

`Reserved`

`Granularity`

A bit mask indicating which bits have been decoded.

`MinimumAddress`

The minimum starting address. For bridges that translate addresses, this is the address space on the secondary side of the bridge.

`MaximumAddress`

The maximum starting address. For bridges that translate addresses, this is the address space on the secondary side of the bridge.

`TranslationAddress`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`AddressLength`

The address length.

`TypeAttribute`

Indicates attributes that are specific to the resource type that is specified in the **ResourceFlags** member. If **ResourceFlags** is zero, this value is zero, otherwise the meaning of the value can be found in the *UEFI Specification* in the section titled **GetMemoryMap()**.

`DescriptorName`

The name of this resource descriptor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

PEP_ACPI_GPIO_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_GPIO_RESOURCE** structure describes the ACPI configuration for a general purpose input/output (GPIO) resource.

Syntax

```
typedef struct _PEP_ACPI_GPIO_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE      Type;
  PEP_ACPI_RESOURCE_FLAGS     Flags;
  KINTERRUPT_MODE             InterruptType;
  KINTERRUPT_POLARITY         InterruptPolarity;
  GPIO_PIN_CONFIG_TYPE        PinConfig;
  GPIO_PIN_IORESTRICTION_TYPE IoRestrictionType;
  USHORT                      DriveStrength;
  USHORT                      DebounceTimeout;
  PUSHORT                     PinTable;
  USHORT                      PinCount;
  UCHAR                       ResourceSourceIndex;
  PUNICODE_STRING             ResourceSourceName;
  PUCHAR                      VendorData;
  USHORT                      VendorDataLength;
} PEP_ACPI_GPIO_RESOURCE, *PPEP_ACPI_GPIO_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value that identifies the resource type for this ACPI resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptPolarity`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`IoRestrictionType`

A GPIO_PIN_IORESTRICTION_TYPE enumeration value that identifies the type of IO that the pin supports.

`DriveStrength`

Specifies the output drive capability of the pin, in hundredths of milliamperes.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`PinTable`

A list of pin numbers on the resource that are described by this descriptor.

`PinCount`

The number of pins in **PinTable**.

`ResourceSourceIndex`

This member is always zero.

`ResourceSourceName`

This member is always set to "ResourceConsumer."

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the **VendorData** buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

GPIO_PIN_IORESTRICTION_TYPE

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_INTERRUPT_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INTERRUPT_RESOURCE** structure describes an ACPI interrupt resource.

Syntax

```
typedef struct _PEP_ACPI_INTERRUPT_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE  Type;
  KINTERRUPT_MODE         InterruptType;
  KINTERRUPT_POLARITY     InterruptPolarity;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  UCHAR                   Count;
  PULONG                  Pins;
} PEP_ACPI_INTERRUPT_RESOURCE, *PPEP_ACPI_INTERRUPT_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptPolarity`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`Count`

The count of items in the **Pins** array.

`Pins`

A list of pin numbers on the resource that are described by this descriptor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_IO_MEMORY_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_IO_MEMORY_RESOURCE** structure describes an ACPI IO port descriptor resource.

Syntax

```
typedef struct _PEP_ACPI_IO_MEMORY_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE Type;
  UCHAR                  Information;
  PHYSICAL_ADDRESS       MinimumAddress;
  PHYSICAL_ADDRESS       MaximumAddress;
  ULONG                  Alignment;
  ULONG                  Length;
} PEP_ACPI_IO_MEMORY_RESOURCE, *PPEP_ACPI_IO_MEMORY_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value that identifies the resource type for this ACPI resource.

`Information`

If bit 0 is a 1, this indicates that the logical device decodes 16-bit addresses. If bit 0 is 0, this indicates that the logical device only decodes the first 10 bits of the address.

Bits 1 to 7 of this member are reserved and must be set to zero.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the IO address assigned.

`Length`

Specifies the number of bytes in the IO range.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_OBJECT_NAME union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_NAME** union contains the four-character name of an ACPI object.

Syntax

```
typedef union _PEP_ACPI_OBJECT_NAME {
  UCHAR Name[4];
  ULONG NameAsUlong;
} PEP_ACPI_OBJECT_NAME, *PPEP_ACPI_OBJECT_NAME;
```

Members

`Name`

`NameAsUlong`

Remarks

The *Name* member of the PEP_ACPI_OBJECT_NAME_WITH_TYPE structure is a **PEP_ACPI_OBJECT_NAME** union. Also, the *Name* member of the PEP_ACPI_QUERY_OBJECT_INFORMATION structure is a **PEP_ACPI_OBJECT_NAME** union.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME_WITH_TYPE

PEP_ACPI_QUERY_OBJECT_INFORMATION

PEP_ACPI_OBJECT_NAME_WITH_TYPE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_NAME_WITH_TYPE** structure that specifies both the path-relative name of an ACPI object and the type of this object.

Syntax

```
typedef struct _PEP_ACPI_OBJECT_NAME_WITH_TYPE {
  PEP_ACPI_OBJECT_NAME Name;
  PEP_ACPI_OBJECT_TYPE Type;
} PEP_ACPI_OBJECT_NAME_WITH_TYPE, *PPEP_ACPI_OBJECT_NAME_WITH_TYPE;
```

Members

`Name`

A PEP_ACPI_OBJECT_NAME structure that contains the four-character name of the ACPI object.

`Type`

A PEP_ACPI_OBJECT_TYPE enumeration value that indicates the type of ACPI object. Currently, the only supported object type is a control method.

Remarks

The **Objects** member of the PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure is the first element in an array of **PEP_ACPI_OBJECT_NAME_WITH_TYPE** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE

PEP_ACPI_OBJECT_NAME

PEP_ACPI_OBJECT_TYPE

PEP_ACPI_OBJECT_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_OBJECT_TYPE** enumeration indicates the type of ACPI object.

Syntax

```
typedef enum _PEP_ACPI_OBJECT_TYPE {
  PepAcpiObjectTypeMethod,
  PepAcpiObjectTypeDevice,
  PepAcpiObjectTypeMaximum
} PEP_ACPI_OBJECT_TYPE, *PPEP_ACPI_OBJECT_TYPE;
```

Constants

PepAcpiObjectTypeMethod	The object is an ACPI control method.
PepAcpiObjectTypeDevice	
PepAcpiObjectTypeMaximum	Reserved for use by the operating system.

Remarks

The **Type** member of the PEP_ACPI_QUERY_OBJECT_INFORMATION structure is an **PEP_ACPI_OBJECT_TYPE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_QUERY_OBJECT_INFORMATION

PEP_ACPI_PREPARE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_PREPARE_DEVICE** structure indicates whether a platform extension plug-in (PEP) is prepared to provide ACPI services for the specified device.

Syntax

```
typedef struct _PEP_ACPI_PREPARE_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  ULONG            InputFlags;
  BOOLEAN          DeviceAccepted;
  ULONG            OutputFlags;
} PEP_ACPI_PREPARE_DEVICE, *PPEP_ACPI_PREPARE_DEVICE;
```

Members

`AcpiDeviceName`

[in] A pointer to an ANSI_STRING structure that contains the fully qualified BIOS name for the device. This name specifies the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_PREPARE_DEVICE_INPUT_FLAG_NONE (0x0).

`DeviceAccepted`

[out] Whether the PEP is prepared to provide ACPI services for the device. Set this member to TRUE if the PEP is prepared to be the sole provider of ACPI services for this device, and to FALSE if it is not.

`OutputFlags`

[out] A set of output flags. No flags are currently defined for this member. Set this member to PEP_ACPI_PREPARE_DEVICE_OUTPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_PREPARE_DEVICE notification. The **AcpiDeviceName** and **InputFlags** members of the structure contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **DeviceAccepted** and **OutputFlags** members contain output values that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_NOTIFY_ACPI_PREPARE_DEVICE

PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES** structure contains a list of raw resources that are needed to control power to the device.

Syntax

```
typedef struct _PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES {
  PEPHANDLE            DeviceHandle;
  ULONG                RequestFlags;
  NTSTATUS             Status;
  SIZE_T               BiosResourcesSize;
  ACPI_METHOD_ARGUMENT BiosResources[ANYSIZE_ARRAY];
} PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES, *PPEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_QDCR_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of this query. Set this member to STATUS_SUCCESS if the PEP succeeded in providing the requested list of resources. Set to STATUS_BUFFER_TOO_SMALL to indicate that the output buffer is not large enough to contain the resource list.

`BiosResourcesSize`

[in, out] On input, the size, in bytes, of the buffer allocated for the **BiosResources** member. This buffer size includes any associated data that might follow this member.

If this buffer is not large enough to contain the **ACPI_METHOD_ARGUMENT** structure and its associated data, the PEP overwrites the input value of **BiosResourcesSize** with the required size and sets the **Status** member to STATUS_BUFFER_TOO_SMALL.

`BiosResources`

[in] An output buffer to which the PEP writes an ACPI_METHOD_ARGUMENT structure that contains the requested resource list. The **Data** member of this structure is the first member in an array that might extend past the end of the structure. The buffer size specified by **BiosResourcesSize** includes both the **ACPI_METHOD_ARGUMENT** structure and any **Data** array elements that follow this structure. For more information, see TBD.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES notification. The **RequestFlags**, **BiosResourcesSize**, and **BiosResources** members of the structure contain input values supplied by the Windows power management framework (PoFx) when this notification is sent. If the **BiosResourcesSize** input value is too small, the PEP should overwrite this value. The **Status** member contains an output value that the PEP writes to this structure in response to the notification.

If the input value specified in the **BiosResourcesSize** member is less than the required size, the PEP overwrites the **BiosResourcesSize** input value with the required size, and sets the **Status** member to STATUS_BUFFER_TOO_SMALL. In response, PoFx will try to allocate a buffer of the required size and send a second **PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES** notification to the PEP for this device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_QUERY_OBJECT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_QUERY_OBJECT_INFORMATION** structure contains information about an ACPI object.

Syntax

```
typedef struct _PEP_ACPI_QUERY_OBJECT_INFORMATION {
  PEPHANDLE            DeviceHandle;
  PEP_ACPI_OBJECT_NAME Name;
  PEP_ACPI_OBJECT_TYPE Type;
  ULONG                ObjectFlags;
  union {
    struct {
      ULONG InputArgumentCount;
      ULONG OutputArgumentCount;
    } MethodObject;
  } DUMMYUNIONNAME;
} PEP_ACPI_QUERY_OBJECT_INFORMATION, *PPEP_ACPI_QUERY_OBJECT_INFORMATION;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`Name`

[in] A PEP_ACPI_OBJECT_NAME union that specifies the path-relative, four-character name of the object.

`Type`

[in] A PEP_ACPI_OBJECT_TYPE enumeration value that specifies the object type. Currently, a control method is the only object type that is defined for this member (**Type** = **PepAcpiObjectTypeMethod**).

`ObjectFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_OBJECT_FLAG_NONE (0x0).

`DUMMYUNIONNAME`

The query result. If the specified object is a control method, the platform extension plug-in (PEP) writes the query result to the **MethodObject** member of this union.

`DUMMYUNIONNAME.MethodObject`

`DUMMYUNIONNAME.MethodObject.InputArgumentCount`

The number of input arguments expected by the control method.

`DUMMYUNIONNAME.MethodObject.OutputArgumentCount`

The number of output arguments produced by the control method.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION notification. The **Name**, **Type**, and **Flags** members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent. The **MethodObject** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_OBJECT_NAME

PEP_ACPI_OBJECT_TYPE

PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_REGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_REGISTER_DEVICE** structure contains registration information about a device for which the platform extension plug-in (PEP) is to provide ACPI services.

Syntax

```
typedef struct _PEP_ACPI_REGISTER_DEVICE {
  PCUNICODE_STRING AcpiDeviceName;
  ULONG            InputFlags;
  POHANDLE         KernelHandle;
  PEPHANDLE        DeviceHandle;
  ULONG            OutputFlags;
} PEP_ACPI_REGISTER_DEVICE, *PPEP_ACPI_REGISTER_DEVICE;
```

Members

`AcpiDeviceName`

[in] An ANSI_STRING structure that contains the fully qualified BIOS name for the device. The same name was previously supplied as an input value in the PEP_ACPI_PREPARE_DEVICE notification for the device. This name specifies the path and name of the device in the ACPI namespace. For more information, see Enumerating Child Devices and Control Methods.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_REGISTER_DEVICE_INPUT_FLAG_NONE (0x0).

`KernelHandle`

[in] A POHANDLE value that represents the registration of the device with the Windows power management framework (PoFx). The PEP can use this handle to identify the device in subsequent communications with PoFx.

`DeviceHandle`

[out] A PEPHANDLE value that identifies the registration of this device for ACPI services. This handle is created by the PEP. In subsequent ACPI service notifications, PoFx will use this handle to identify the device.

`OutputFlags`

[out] A set of output flags. No flags are currently defined for this member. Set this member to PEP_ACPI_REGISTER_DEVICE_OUTPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification. The **AcpiDeviceName**, **InputFlags**, and **KernelHandle** members of the structure contain input values that are supplied by PoFx when this notification is sent to the PEP. The **DeviceHandle** and **OutputFlags** members contains output values that the PEP writes to this structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ANSI_STRING

PEP_ACPI_PREPARE_DEVICE

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES** structure is used in the process of converting ACPI resources to BIOS resources by one of the PEP initialization functions.

Syntax

```
typedef struct _PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES {
  NTSTATUS           TranslationStatus;
  PPEP_ACPI_RESOURCE InputBuffer;
  SIZE_T             InputBufferSize;
  PVOID              OutputBuffer;
  SIZE_T             OutputBufferSize;
  ULONG              Flags;
} PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES, *PPEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES;
```

Members

`TranslationStatus`

The result code from the resource translation call.

`InputBuffer`

A pointer to a PEP_ACPI_RESOURCE containing the input buffer.

`InputBufferSize`

The size of the input buffer.

`OutputBuffer`

A pointer to the output buffer containing the translated structure.

`OutputBufferSize`

The size of the output buffer.

`Flags`

The value contained in the PEP_ACPI_RESOURCE_FLAGS structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP initialization functions

PEP_ACPI_RESOURCE

PEP_ACPI_RESOURCE_FLAGS

PEP_ACPI_RESOURCE union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE** structure contains hardware details for a specific ACPI resource.

Syntax

```
typedef union _PEP_ACPI_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE      Type;
  PEP_ACPI_IO_MEMORY_RESOURCE IoMemory;
  PEP_ACPI_INTERRUPT_RESOURCE Interrupt;
  PEP_ACPI_GPIO_RESOURCE      Gpio;
  PEP_ACPI_SPB_I2C_RESOURCE   SpbI2c;
  PEP_ACPI_SPB_SPI_RESOURCE   SpbSpi;
  PEP_ACPI_SPB_UART_RESOURCE  SpbUart;
  PEP_ACPI_EXTENDED_ADDRESS   ExtendedAddress;
} PEP_ACPI_RESOURCE, *PPEP_ACPI_RESOURCE;
```

Members

`Type`

The PEP_ACPI_RESOURCE_TYPE enumeration value that is applicable to this resource.

`IoMemory`

If **Type** is **PepAcpiMemory** or **PepAcpiIoPort**, this union contains a PEP_ACPI_IO_MEMORY_RESOURCE structure describing an ACPI IO or memory resource.

`Interrupt`

If **Type** is **PepAcpiInterrupt**, this union contains a PEP_ACPI_INTERRUPT_RESOURCE structure describing an ACPI interrupt resource.

`Gpio`

If **Type** is **PepAcpiGpioIo** or **PepAcpiGpioInt**, this union contains a PEP_ACPI_GPIO_RESOURCE structure describing an ACPI GPIO resource.

`SpbI2c`

If **Type** is **PepAcpiSpbI2c**, this union contains a PEP_ACPI_SPB_I2C_RESOURCE structure describing an ACPI I2C serial bus resource.

`SpbSpi`

If **Type** is **PepAcpiSpbSpi**, this union contains a PEP_ACPI_SPB_SPI_RESOURCE structure describing an ACPI SPI serial bus resource.

`SpbUart`

If **Type** is **PepAcpiSpbUart**, this union contains a PEP_ACPI_SPB_UART_RESOURCE structure describing an ACPI SPB serial bus resource.

`ExtendedAddress`

If **Type** is **PepAcpiExtendedMemory** or **PepAcpiExtendedIo**, this union contains a PEP_ACPI_EXTENDED_ADDRESS structure describing an ACPI extended memory or extended IO resource.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INTERRUPT_RESOURCE

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_RESOURCE_TYPE

PEP_ACPI_SPB_I2C_RESOURCE

PEP_ACPI_SPB_SPI_RESOURCE

PEP_ACPI_SPB_UART_RESOURCE

PEP_ACPI_RESOURCE_FLAGS union

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE_FLAGS** structure contains flags describing an ACPI resource.

Syntax

```
typedef union _PEP_ACPI_RESOURCE_FLAGS {
  ULONG  AsULong;
  struct {
    ULONG Shared : 1;
    ULONG Wake : 1;
    ULONG ResourceUsage : 1;
    ULONG SlaveMode : 1;
    ULONG AddressingMode : 1;
    ULONG SharedMode : 1;
    ULONG Reserved : 26;
  } DUMMYSTRUCTNAME;
} PEP_ACPI_RESOURCE_FLAGS, *PPEP_ACPI_RESOURCE_FLAGS;
```

Members

`AsULong`

The consolidated values of the flags in **DUMMYSTRUCTNAME**.

`DUMMYSTRUCTNAME`

A structure containing ACPI resource flags.

`DUMMYSTRUCTNAME.Shared`

When set, indicates that this is a shared device.

`DUMMYSTRUCTNAME.Wake`

When set, indicates that this device can be woken from a low-power state.

`DUMMYSTRUCTNAME.ResourceUsage`

When set, indicates that this device is in use.

`DUMMYSTRUCTNAME.SlaveMode`

When set, indicates that this device is in slave mode.

`DUMMYSTRUCTNAME.AddressingMode`

When set, indicates that this device is in addressing mode.

`DUMMYSTRUCTNAME.SharedMode`

When set, indicates that this device is in shared mode.

`DUMMYSTRUCTNAME.Reserved`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

PEP_ACPI_RESOURCE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_RESOURCE_TYPE** enumeration is used to identify the type of ACPI resource that is contained in the PEP_ACPI_RESOURCE union.

Syntax

```
typedef enum _PEP_ACPI_RESOURCE_TYPE {
  PepAcpiMemory,
  PepAcpiIoPort,
  PepAcpiInterrupt,
  PepAcpiGpioIo,
  PepAcpiGpioInt,
  PepAcpiSpbI2c,
  PepAcpiSpbSpi,
  PepAcpiSpbUart,
  PepAcpiExtendedMemory,
  PepAcpiExtendedIo
} PEP_ACPI_RESOURCE_TYPE;
```

Constants

PepAcpiMemory	Indicates that the resource is an ACPI memory resource.
PepAcpiIoPort	Indicates that the resource is an ACPI IO port resource.
PepAcpiInterrupt	Indicates that the resource is an ACPI interrupt resource.
PepAcpiGpioIo	Indicates that the resource is an ACPI GPIO resource.
PepAcpiGpioInt	Indicates that the resource is an ACPI GPIO interrupt resource.
PepAcpiSpbI2c	Indicates that the resource is an ACPI I2C serial bus resource.
PepAcpiSpbSpi	Indicates that the resource is an ACPI SPI serial bus resource.
PepAcpiSpbUart	Indicates that the resource is an ACPI UART serial bus resource.
PepAcpiExtendedMemory	Indicates that the resource is an ACPI extended memory resource.
PepAcpiExtendedIo	Indicates that the resource is an ACPI extended IO resource.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_REQUEST_CONVERT_TO_BIOS_RESOURCES

PEP_ACPI_RESOURCE

PEP_ACPI_SPB_I2C_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_I2C_RESOURCE** structure describes an ACPI I2C serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_I2C_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 ConnectionSpeed;
  USHORT                SlaveAddress;
} PEP_ACPI_SPB_I2C_RESOURCE, *PPEP_ACPI_SPB_I2C_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`SlaveAddress`

The I2C bus address for this connection.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_RESOURCE

PEP_ACPI_SPB_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_RESOURCE** structure describes an ACPI serial bus connection resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_RESOURCE {
  PEP_ACPI_RESOURCE_TYPE  Type;
  PEP_ACPI_RESOURCE_FLAGS Flags;
  USHORT                  TypeSpecificFlags;
  UCHAR                   ResourceSourceIndex;
  PUNICODE_STRING         ResourceSourceName;
  PCHAR                   VendorData;
  USHORT                  VendorDataLength;
} PEP_ACPI_SPB_RESOURCE, *PPEP_ACPI_SPB_RESOURCE;
```

Members

`Type`

A PEP_ACPI_RESOURCE_TYPE enumeration value describing this resource.

`Flags`

A PEP_ACPI_RESOURCE_FLAGS structure that describes the capabilities of this ACPI resource.

`TypeSpecificFlags`

Specifies the bit flags that are common to all serial bus connection types.

BIT(S)	MEANING
0 - Slave mode	**0x0** - Indicates that communication over this connection is initiated by the controller. **0x1** - Indicates that communication over this connection is initiated by the device.
1 - Consumer/producer mode	**0x0** - Indicates that this device produces and consumes this resource. **0x1** - Indicates that this device consumes this resource.

2 to 7 - Reserved	These bits are reserved and must be set to zero.

`ResourceSourceIndex`

`ResourceSourceName`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by **VendorData**.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

PEP_ACPI_SPB_SPI_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_SPI_RESOURCE** structure describes an ACPI SPI serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_SPI_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 ConnectionSpeed;
  UCHAR                 DataBitLength;
  UCHAR                 Phase;
  UCHAR                 Polarity;
  USHORT                DeviceSelection;
} PEP_ACPI_SPB_SPI_RESOURCE, *PPEP_ACPI_SPB_SPI_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`DataBitLength`

The size, in bits, of the smallest unit of transfer.

`Phase`

The phase of the clock pulse on which to capture data.

`Polarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`DeviceSelection`

The device selection value. This value is specific to the device and may refer to a chip-select line, GPIO line, or other line selection mechanism.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_UART_RESOURCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_SPB_UART_RESOURCE** structure describes an ACPI UART serial bus resource.

Syntax

```
typedef struct _PEP_ACPI_SPB_UART_RESOURCE {
  PEP_ACPI_SPB_RESOURCE SpbCommon;
  ULONG                 BaudRate;
  USHORT                RxBufferSize;
  USHORT                TxBufferSize;
  UCHAR                 Parity;
  UCHAR                 LinesInUse;
} PEP_ACPI_SPB_UART_RESOURCE, *PPEP_ACPI_SPB_UART_RESOURCE;
```

Members

`SpbCommon`

A PEP_ACPI_SPB_RESOURCE structure describing this resource.

`BaudRate`

The baud rate of the connection.

`RxBufferSize`

The maximum receive buffer size, in bytes, that is supported by this connection.

`TxBufferSize`

The maximum transmit buffer size, in bytes, that is supported by this connection.

`Parity`

Indicates the parity of the connection.

VALUE	MEANING
0x0 0	None
0x0 1	Even
0x0 2	Odd

0x03	Mark
0x04	Space

`LinesInUse`

Flag indicating the serial lines that are enabled. A value of 1 in the bit positions indicates that the line is enabled.

BIT	MEANING
0	This bit is reserved and must be set to zero.
1	This bit is reserved and must be set to zero.
2	Data Carrier Detect (DTD)
3	Ring Indicator (RI)
4	Data Set Ready (DSR)
5	Data Terminal Ready (DTR)
6	Clear to Send (CTS)
7	Request to Send (RTS)

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_RESOURCE

PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES** structure contains a list of translated power-control resources for the platform extension plug-in (PEP) to use.

Syntax

```
typedef struct _PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES {
  PEPHANDLE         DeviceHandle;
  ULONG             RequestFlags;
  NTSTATUS          Status;
  SIZE_T            TranslatedResourcesSize;
  PCM_RESOURCE_LIST TranslatedResources;
} PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES, *PPEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`RequestFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_TDCR_FLAG_NONE (0x0).

`Status`

[out] An NTSTATUS value that indicates the status of the resource translation. The PEP sets this member to STATUS_SUCCESS to indicate that the PEP successfully received the translated resources. Otherwise, the PEP sets this member to an appropriate error status code.

`TranslatedResourcesSize`

`TranslatedResources`

[in] A CM_RESOURCE_LIST structure that serves as the header for the resource list. The remainder of the resource list immediately follows this header.

Remarks

This structure is used by the PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification to provide the PEP with a list of translated power control resources. The **RequestFlags**, **ResourceBufferSize**, and **TranslatedResources** members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent. The **Status** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

CM_RESOURCE_LIST

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES

PEP_ACPI_UNREGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_UNREGISTER_DEVICE** structure contains information about a device that has been unregistered from ACPI services.

Syntax

```
typedef struct _PEP_ACPI_UNREGISTER_DEVICE {
  PEPHANDLE DeviceHandle;
  ULONG     InputFlags;
} PEP_ACPI_UNREGISTER_DEVICE, *PPEP_ACPI_UNREGISTER_DEVICE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device's registration for ACPI services. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

`InputFlags`

[in] A set of input flags. No flags are currently defined for this member, which is always set to PEP_ACPI_UNREGISTER_DEVICE_INPUT_FLAG_NONE (0x0).

Remarks

This structure is used by the PEP_NOTIFY_ACPI_UNREGISTER_DEVICE notification. The **DeviceHandle** and **InputFlags** members contain input values that are supplied by the Windows power management framework when this notification is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_NOTIFY_ACPI_UNREGISTER_DEVICE

PEP_COMPONENT_PERF_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_INFO** structure describes the performance states (P-states) of a component.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_INFO {
  ULONG                  SetCount;
  PEP_COMPONENT_PERF_SET PerfStateSets[ANYSIZE_ARRAY];
} PEP_COMPONENT_PERF_INFO, *PPEP_COMPONENT_PERF_INFO;
```

Members

`SetCount`

The number of elements in the **PerfStateSets** array.

`PerfStateSets`

The first element in an array of PEP_COMPONENT_PERF_SET structures. Each array element describes a P-state of the component. If the array contains more than one element, the additional elements immediately follow the end of the **PEP_COMPONENT_PERF_INFO** structure.

Remarks

This structure is used in conjunction with the PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification. The **PerfStateSets** member of the PEP_REGISTER_COMPONENT_PERF_STATES structure is a **PEP_COMPONENT_PERF_INFO** structure.

The **PEP_COMPONENT_PERF_INFO** structure contains input values that are supplied by the Windows power management framework (PoFx). The platform extension plug-in (PEP) must not write to this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_COMPONENT_PERF_SET

PEP_DPM_REGISTER_COMPONENT_PERF_STATES

PEP_REGISTER_COMPONENT_PERF_STATES

PEP_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_SET** structure describes the performance states (P-states) in a P-state set.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_SET {
  UNICODE_STRING      Name;
  ULONGLONG           Flags;
  PEP_PERF_STATE_UNIT Unit;
  PEP_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG           Count;
      PPEP_PERF_STATE States;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PEP_COMPONENT_PERF_SET, *PPEP_COMPONENT_PERF_SET;
```

Members

`Name`

An optional string that describes the device property controlled by this P-state set. For example, this string might be "Clock frequency" or "Memory bandwidth". If no such string is available for this P-state, the **Name** member is set to NULL. Otherwise, this member contains a pointer to a UNICODE_STRING structure that contains the string.

`Flags`

Set to zero. No flag bits are currently defined for this member.

`Unit`

A PEP_PERF_STATE_UNIT structure that specifies the units in which the performance values for this P-state set are expressed. Component performance can be expressed in hertz (frequency) or in bits per second (bandwidth).

`Type`

A PEP_PERF_STATE_TYPE enumeration value that indicates the type of performance information that is specified for this component. This member indicates whether the performance values for this P-state set are specified as a list of discrete values or as a continuous range of values.

`Discrete`

`Discrete.Count`

`Discrete.States`

`Range`

`Range.Minimum`

`Range.Maximum`

Remarks

The **PerfStateSets** member of the PEP_COMPONENT_PERF_INFO is the first element in an array of **PEP_COMPONENT_PERF_SET** structures. All members of the **PEP_COMPONENT_PERF_SET** structure contain input values that are supplied by the Windows power management framework (PoFx). The platform extension plug-in (PEP) must not write to this structure.

Device drivers use the PO_FX_COMPONENT_PERF_SET structure, which is similar to the **PEP_COMPONENT_PERF_SET** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_PERF_STATE

PEP_PERF_STATE_TYPE

PEP_PERF_STATE_UNIT

PO_FX_COMPONENT_PERF_SET

UNICODE_STRING

PEP_COMPONENT_PERF_STATE_REQUEST structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PERF_STATE_REQUEST** structure specifies a performance state (P-state) set and a new performance level to assign to this set.

Syntax

```
typedef struct _PEP_COMPONENT_PERF_STATE_REQUEST {
  ULONG Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PEP_COMPONENT_PERF_STATE_REQUEST, *PPEP_COMPONENT_PERF_STATE_REQUEST;
```

Members

`Set`

The index of the P-state set to which to assign the new performance level. If N is the number of P-state sets specified for this component, P-state set indexes range from 0 to N−1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`StateIndex`

`StateValue`

Remarks

The **PerfRequests** member of the PEP_REQUEST_COMPONENT_PERF_STATE structure is a pointer to an array of **PEP_COMPONENT_PERF_STATE_REQUEST** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_REQUEST_COMPONENT_PERF_STATE

PEP_COMPONENT_PLATFORM_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_PLATFORM_CONSTRAINTS** structure describes the lowest-powered F*x* state of that a component can be in when the platform is in a particular idle state.

Syntax

```
typedef struct _PEP_COMPONENT_PLATFORM_CONSTRAINTS {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  PULONG    MinimumFStates;
  ULONG     PlatformStateCount;
} PEP_COMPONENT_PLATFORM_CONSTRAINTS, *PPEP_COMPONENT_PLATFORM_CONSTRAINTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`MinimumFStates`

[in] A pointer to an output buffer. The PEP writes a ULONG array to this buffer that specifies the lowest-powered F*x* state that the component can be in for each platform idle state. An element with a value of 0 indicates F0, a value of 1 indicates F1, and so on. If the platform supports M idle states, array elements 0 to M−1 specify the F*x* states corresponding to platform idle states 0 to M−1.

`PlatformStateCount`

[in] The number of elements in the **MinimumFStates** array. The array contains one element for each platform idle state. The Windows power management framework (PoFx) previously sent a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification to the PEP to determine the number of supported platform idle states.

Remarks

This structure is used by the PEP_DPM_COMPONENT_IDLE_CONSTRAINTS notification. All four members of the structure contain input values that PoFx supplies when this notification is sent. PoFx allocates the buffer pointed to by the **MinimumFStates** member, and the PEP writes to this buffer in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_COMPONENT_IDLE_CONSTRAINTS

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_COMPONENT_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COMPONENT_V2** structure specifies the power state attributes of a component in the device.

Syntax

```
typedef struct _PEP_COMPONENT_V2 {
  GUID                          Id;
  ULONGLONG                     Flags;
  ULONG                         DeepestWakeableIdleState;
  ULONG                         IdleStateCount;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
} PEP_COMPONENT_V2, *PPEP_COMPONENT_V2;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The PEP should specify a nonzero value for this member if the Windows power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`Flags`

A set of component-power-state flags. No flags are currently defined for this member, which is always zero.

`DeepestWakeableIdleState`

The index of the deepest F*x* state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of F*x* power states that the component supports. A component must support at least one F*x* state (F0).

`IdleStates`

A pointer to an array of PO_FX_COMPONENT_IDLE_STATE structures. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an F*x* power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

Remarks

This structure

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PO_FX_COMPONENT_IDLE_STATE

PEP_COORDINATED_DEPENDENCY_OPTION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COORIDNATED_DEPENDENCY_OPTION** structure describes a coordinated idle state's dependency to the OS.

Syntax

```
typedef struct _PEP_COORDINATED_DEPENDENCY_OPTION {
  UCHAR   ExpectedStateIndex;
  BOOLEAN LooseDependency;
  BOOLEAN InitiatingState;
  BOOLEAN DependentState;
} PEP_COORDINATED_DEPENDENCY_OPTION, *PPEP_COORDINATED_DEPENDENCY_OPTION;
```

Members

`ExpectedStateIndex`

Supplies the index of the idle state (coordinated idle state or processor idle state) that must have already been entered/be entered by this idle transition.

`LooseDependency`

When **FALSE**, the OS must use synchronization to guarantee that the state referred to by this dependency has already been entered. When **TRUE**, the OS may perform best-effort coordination that the target state is entered. The PEP must set **LooseDependency** to **TRUE** if the **ExpectedState** is a processor state and has the **WakesSpuriously** flag set.

`InitiatingState`

Indicates if the OS may simultaneously enter the state referred to by this dependency. If **FALSE**, this state is only valid when another processor has previously entered it.

`DependentState`

Indicates if this state is a valid state for a dependent processor to be in.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY notification

PEP_COORDINATED_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_COORIDNATED_IDLE_STATE** structure describes a coordinated idle state to the OS.

Syntax

```
typedef struct _PEP_COORDINATED_IDLE_STATE {
  ULONG Latency;
  ULONG BreakEvenDuration;
  ULONG DependencyCount;
  ULONG MaximumDependencySize;
} PEP_COORDINATED_IDLE_STATE, *PPEP_COORDINATED_IDLE_STATE;
```

Members

`Latency`

The latency of waking from this idle state, in 100ns units.

`BreakEvenDuration`

Supplies the minimum time the state must be entered to amortize the cost of entering/exiting the state. Idle durations longer than this period should save power when compared to entering a lighter state for the same period.

`DependencyCount`

Supplies the number of dependencies this coordinated state has on other coordinated states or on processors.

`MaximumDependencySize`

Supplies the maximum size of a single dependency.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COORDINATED_IDLE_STATE structure

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification

PEP_CRASHDUMP_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_CRASHDUMP_INFORMATION** structure contains information about a crash-dump device.

Syntax

```
typedef struct _PEP_CRASHDUMP_INFORMATION {
  PEPHANDLE DeviceHandle;
  PVOID     DeviceContext;
} PEP_CRASHDUMP_INFORMATION, *PPEP_CRASHDUMP_INFORMATION;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the crash-dump device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`DeviceContext`

[in] A pointer to a device-specific context. This pointer is the *Context* parameter value that the device driver previously passed to the PoFxPowerOnCrashdumpDevice routine. The context contains information in a format that is defined by the device driver and is understood by the PEP. The context is opaque to the operating system. This member value can be NULL if the PEP does not require a context.

Remarks

The *CrashdumpInformation* parameter to the PowerOnDumpDeviceCallback routine is a pointer to a **PEP_CRASHDUMP_INFORMATION** structure.

When a fatal system error occurs, the devices in the crash-dump device chain (storage controller, PCI controller, and so on) need to be turned on so that the Windows kernel can write a crash-dump file to disk.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PoFxPowerOnCrashdumpDevice

PowerOnDumpDeviceCallback

PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEBUGGER_TRANSITION_REQUIREMENTS** structure indicates the platform idle states for which the debugger device must be turned on.

Syntax

```
typedef struct _PEP_DEBUGGER_TRANSITION_REQUIREMENTS {
  PEPHANDLE DeviceHandle;
  ULONG     PlatformStateCount;
  PBOOLEAN  PowerTransitionRequired;
} PEP_DEBUGGER_TRANSITION_REQUIREMENTS, *PPEP_DEBUGGER_TRANSITION_REQUIREMENTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the debugger device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PlatformStateCount`

[in] The number of entries in the array pointed to by the **PowerTransitionRequired** member. This number is the same as the number of supported platform idle states that the PEP specified in response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

`PowerTransitionRequired`

[in] An output buffer to which the PEP writes a BOOLEAN array that indicates the platform idle states for which power to the debugger device must be turned on. This buffer is allocated by the Windows power management framework (PoFx) and is guaranteed to be large enough to contain the number of array elements specified by the **PlatformStateCount** member. Set an array element to TRUE if the debugger device must be turned on in the corresponding platform idle state. Set to FALSE to indicate that the device is not required to be turned on in this state. If the platform supports N idle states, the array contains N elements, and idle states are numbered 0 to N–1 in the order in which they are represented in the array.

Remarks

This structure is used by the PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS notification. All three members of the structure contain input values that are supplied by PoFx when this notification is sent to the PEP.

During a debugging session, the debugger component that runs on the target computer communicates with the host computer through a debugger device such as a USB host controller, UART, or networking device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_DEVICE_ACCEPTANCE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_ACCEPTANCE_TYPE** enumeration indicates whether a PEP accepts ownership of a device.

Syntax

```
typedef enum _PEP_DEVICE_ACCEPTANCE_TYPE {
  PepDeviceNotAccepted,
  PepDeviceAccepted,
  PepDeviceAceptedMax
} PEP_DEVICE_ACCEPTANCE_TYPE, *PPEP_DEVICE_ACCEPTANCE_TYPE;
```

Constants

PepDeviceNotAccepted	The PEP does not claim ownership of this device.
PepDeviceAccepted	The PEP claims ownership of this device.
PepDeviceAceptedMax	

Remarks

This enumeration is used by **DeviceAccepted** member of the PEP_REGISTER_DEVICE_V2 structure to indicate whether a PEP accepts ownership of a device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_REGISTER_DEVICE_V2

PEP_DEVICE_PLATFORM_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_PLATFORM_CONSTRAINTS** structure specifies the constraints for entry to the various D*x* power states that are supported by a device.

Syntax

```
typedef struct _PEP_DEVICE_PLATFORM_CONSTRAINTS {
  PEPHANDLE           DeviceHandle;
  PDEVICE_POWER_STATE MinimumDStates;
  ULONG               PlatformStateCount;
} PEP_DEVICE_PLATFORM_CONSTRAINTS, *PPEP_DEVICE_PLATFORM_CONSTRAINTS;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`MinimumDStates`

[in] A pointer to an array of DEVICE_POWER_STATE enumeration values that indicate the lowest-powered D*x* (device power) state the device can enter for each platform idle state.

`PlatformStateCount`

[in] The number of elements in the array pointed to by the **MinimumDStates** member. This member contains the platform state count that the PEP supplied in response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

Remarks

This structure is used by the PEP_DPM_DEVICE_IDLE_CONSTRAINTS notification. All three members of this structure contain input values that are supplied by the Windows power management framework (PoFx). In response to this notification, the PEP writes DEVICE_POWER_STATE enumeration values to the elements of the array pointed to by the **MinimumDStates** member. PoFx allocates the storage for this array before sending the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

DEVICE_POWER_STATE

PEP_DPM_DEVICE_IDLE_CONSTRAINTS

PEP_DPM_REGISTER_DEVICE

PEP_DEVICE_POWER_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_POWER_STATE** structure indicates the status of a transition to a new D*x* (device power) state.

Syntax

```
typedef struct _PEP_DEVICE_POWER_STATE {
  PEPHANDLE          DeviceHandle;
  DEVICE_POWER_STATE PowerState;
  BOOLEAN            Complete;
  BOOLEAN            SystemTransition;
} PEP_DEVICE_POWER_STATE, *PPEP_DEVICE_POWER_STATE;
```

Members

`DeviceHandle`

[in] The PEPHANDLE value that identifies this device. The PEP previously created this handle in response to a PEP_DPM_REGISTER_DEVICE notification from the Windows power management framework (PoFx).

`PowerState`

[in] A DEVICE_POWER_STATE enumeration value that specifies the new device power state.

`Complete`

[in] Whether the transition to the new device power state has just been initiated or has just completed. If TRUE, the transition to the target device power state has completed. If FALSE, the power policy owner (PPO) has initiated the transition by calling the PoRequestPowerIrp routine, but the Windows power manager has not yet issued the D*x* IRP (an IRP_MN_SET_POWER request of type **DevicePowerState**) to the device's driver stack.

`SystemTransition`

[in] Always set to FALSE.

Remarks

This structure is used by the PEP_DPM_DEVICE_POWER_STATE notification. All four members of the structure contain input values that are supplied by PoFx. The PEP does not write to this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

DEVICE_POWER_STATE

IRP_MN_SET_POWER

PEP_DPM_DEVICE_POWER_STATE

PEP_DPM_REGISTER_DEVICE

PoRequestPowerIrp

PEP_DEVICE_REGISTER_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_REGISTER** structure describes all the components in a particular device.

Syntax

```
typedef struct _PEP_DEVICE_REGISTER_V2 {
  ULONGLONG         Flags;
  ULONG             ComponentCount;
  PPEP_COMPONENT_V2 Components[ANYSIZE_ARRAY];
} PEP_DEVICE_REGISTER_V2, *PPEP_DEVICE_REGISTER_V2;
```

Members

`Flags`

TBD

`ComponentCount`

The number of components in this device, which is also the number of elements in the **Components** array. The **ComponentCount** value must be greater than or equal to one.

`Components`

The first element in an array of pointers to PEP_COMPONENT_V2 structures. Each element in the array points to a structure that describes a component in the device. If this array contains more than one element, the additional elements immediately follow the end of the **PEP_DEVICE_REGISTER** structure.

Remarks

The PEP_REGISTER_DEVICE structure contains a **Register** member that points to a **PEP_DEVICE_REGISTER** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_V2

PEP_REGISTER_DEVICE

PEP_DEVICE_STARTED structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_DEVICE_STARTED** structure identifies a device whose driver has completed its registration with the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_DEVICE_STARTED {
  PEPHANDLE DeviceHandle;
} PEP_DEVICE_STARTED, *PPEP_DEVICE_STARTED;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_DEVICE_STARTED notification. The **DeviceHandle** member of this structure contains an input value that is supplied by PoFx.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_DEVICE_STARTED

PEP_DPM_REGISTER_DEVICE

PEP_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_INFORMATION** structure specifies the interface that the platform extension plug-in (PEP) uses to receive notifications from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_INFORMATION {
  USHORT                 Version;
  USHORT                 Size;
  PPEPCALLBACKNOTIFYDPM  AcceptDeviceNotification;
  PPEPCALLBACKNOTIFYPPM  AcceptProcessorNotification;
  PPEPCALLBACKNOTIFYACPI AcceptAcpiNotification;
} PEP_INFORMATION, *PPEP_INFORMATION;
```

Members

`Version`

The current version number for this structure. Set this member to PEP_INFORMATION_VERSION.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_INFORMATION**).

`AcceptDeviceNotification`

A pointer to an AcceptDeviceNotification callback routine. This member is required to point to a valid callback routine.

`AcceptProcessorNotification`

A pointer to an AcceptProcessorNotification callback routine. This member is optional and can be NULL if the PEP is not prepared to handle PPM notifications from PoFx.

`AcceptAcpiNotification`

A pointer to an AcceptAcpiNotification callback routine. This member is optional and can be NULL if the PEP is not prepared to handle ACPI notifications from PoFx.

Remarks

This structure contains pointers to several callback routines that are implemented by the PEP. PoFx calls these routines to send notifications to the PEP.

The *PepInformation* parameter to the PoFxRegisterPlugin and PoFxRegisterPluginEx routines is a pointer to a **PEP_INFORMATION** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

AcceptAcpiNotification

AcceptDeviceNotification

AcceptProcessorNotification

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEP_KERNEL_INFORMATION_STRUCT_V3 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_KERNEL_INFORMATION_STRUCT_V3** structure specifies the interface that the power extension plug-in (PEP) uses to request services from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_KERNEL_INFORMATION_STRUCT_V3 {
  USHORT                                  Version;
  USHORT                                  Size;
  POHANDLE                                Plugin;
  PPOFXCALLBACKREQUESTWORKER              RequestWorker;
  PPOFXCALLBACKENUMERATEUNMASKEDINTERRUPTS EnumerateUnmaskedInterrupts;
  PPOFXCALLBACKPROCESSORHALT              ProcessorHalt;
  PPOFXCALLBACKREQUESTINTERRUPT           RequestInterrupt;
  PPOFXCALLBACKCRITICALRESOURCE           TransitionCriticalResource;
  PPOFXCALLBACKPROCESSORIDLEVETO          ProcessorIdleVeto;
  PPOFXCALLBACKPLATFORMIDLEVETO           PlatformIdleVeto;
  PPOFXCALLBACKUPDATEPROCESSORIDLESTATE   UpdateProcessorIdleState;
  PPOFXCALLBACKUPDATEPLATFORMIDLESTATE    UpdatePlatformIdleState;
  PPOFXCALLBACKREQUESTCOMMON              RequestCommon;
} PEP_KERNEL_INFORMATION_STRUCT_V3, *PPEP_KERNEL_INFORMATION_STRUCT_V3;
```

Members

`Version`

The current version number for this structure. Set this member to **PEP_KERNEL_INFORMATION_VERSION**.

`Size`

The size, in bytes, of this structure. Set this member to **sizeof**(**PEP_KERNEL_INFORMATION**).

`Plugin`

The handle assigned to the PEP's registration with PoFx. PoFx sets the value of this member. The PEP uses this handle in calls to the RequestWorker routine.

`RequestWorker`

A pointer to a RequestWorker routine. PoFx sets the value of this member. The **RequestWorker** routine is implemented by PoFx. The PEP calls this routine to request the use of a worker thread from the operating system.

`EnumerateUnmaskedInterrupts`

A pointer to an EnumerateUnmaskedInterrupts routine. PoFx sets the value of this member. The **EnumerateUnmaskedInterrupts** routine is implemented by PoFx. The PEP calls this routine to request information about the unmasked interrupts.

`ProcessorHalt`

A pointer to a ProcessorHalt routine. PoFx sets the value of this member. The **ProcessorHalt** routine is implemented by PoFx. The PEP calls this routine to prepare the current processor to enter the halted state.

`RequestInterrupt`

A pointer to a RequestInterrupt routine. PoFx sets the value of this member. The **RequestInterrupt** routine is implemented by PoFx. The PEP calls this routine to replay an edge-triggered interrupt that might have been lost after the hardware platform entered a low-power system state.

`TransitionCriticalResource`

A pointer to a TransitionCriticalResource routine. PoFx sets the value of this member. The **TransitionCriticalResource** routine is implemented by PoFx. The PEP calls this routine to transition critical system resources to the idle condition.

`ProcessorIdleVeto`

A pointer to a ProcessorIdleVeto routine. PoFx sets the value of this member. The **ProcessorIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a processor idle state.

`PlatformIdleVeto`

A pointer to a PlatformIdleVeto routine. PoFx sets the value of this member. The **PlatformIdleVeto** routine is implemented by PoFx. The PEP calls this routine to increment or decrement the veto count for a pending transition to a platform idle state.

`UpdateProcessorIdleState`

A pointer to an UpdateProcessorIdleState routine. PoFx sets the value of this member. The **UpdateProcessorIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified processor idle state.

`UpdatePlatformIdleState`

A pointer to an UpdatePlatformIdleState routine. PoFx sets the value of this member. The **UpdatePlatformIdleState** routine is implemented by PoFx. The PEP calls this routine to update the properties of the specified platform idle state.

`RequestCommon`

A pointer to an RequestCommon routine. PoFx sets the value of this member. The **RequestCommon** routine is implemented by PoFx. The PEP calls this routine to perform a processing operation that is specified by a request ID.

Remarks

The **KernelInformation** parameter to the PoFxRegisterPlugin or PoFxRegisterPluginEx routine is a pointer to a **PEP_KERNEL_INFORMATION** structure. The PEP allocates this structure and sets the values of the **Version** and **Size** members of this structure before calling **PoFxRegisterPlugin** or **PoFxRegisterPluginEx**. During the call to this routine, PoFx fills in the remaining members of the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

EnumerateUnmaskedInterrupts

PlatformIdleVeto

PoFxRegisterPlugin

PoFxRegisterPluginEx

ProcessorHalt

ProcessorIdleVeto

RequestCommon

RequestInterrupt

RequestWorker

TransitionCriticalResource

UpdatePlatformIdleState

UpdateProcessorIdleState

PEP_LOW_POWER_EPOCH structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_LOW_POWER_EPOCH** structure is used to provide data for a PEP_DPM_LOW_POWER_EPOCH notification (deprecated).

Syntax

```
typedef struct _PEP_LOW_POWER_EPOCH {
  BOOLEAN LowPowerEpoch;
} PEP_LOW_POWER_EPOCH, *PPEP_LOW_POWER_EPOCH;
```

Members

`LowPowerEpoch`

When **TRUE**, indicates that the low power epoch is on.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_LOW_POWER_EPOCH notification

PEP_NOTIFY_COMPONENT_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_NOTIFY_COMPONENT_IDLE_STATE** structure contains status information about a component's pending transition to a new F*x* power state.

Syntax

```
typedef struct _PEP_NOTIFY_COMPONENT_IDLE_STATE {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     IdleState;
  BOOLEAN   DriverNotified;
  BOOLEAN   Completed;
} PEP_NOTIFY_COMPONENT_IDLE_STATE, *PPEP_NOTIFY_COMPONENT_IDLE_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`IdleState`

[in] The new F*x* power state that the component will transition to. If this parameter is zero, the new state is F0; if this parameter is one, the new state is F1; and so on.

`DriverNotified`

[in] Whether the driver for the device has already been notified of the pending transition to the new F*x* state. If TRUE, the power management framework (PoFx) already notified the driver by calling the driver's ComponentIdleStateCallback callback routine. If FALSE, PoFx has not yet notified the driver.

`Completed`

[out] Whether the PEP has completed all necessary preparations for the specified component to enter the new F*x* state. Set to TRUE to indicate that the PEP has completed all necessary preparations, or to FALSE to indicate that is has not.

If FALSE, the PEP must complete the necessary preparations at a later time and then inform PoFx when the preparations have been completed. To do so, the PEP calls the RequestWorker routine to request a work item, and then responds to the ensuing PEP_DPM_WORK notification by submitting a work request of type **PepWorkCompleteIdleState** to inform PoFx that the preparations are complete.

Remarks

This structure is used by the PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE notification. The first four members of this structure contain input values that are supplied by PoFx. The **Completed** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE

PEP_DPM_REGISTER_DEVICE

PEP_DPM_WORK

RequestWorker

PEP_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE** structure describes a performance state (P-state) in a P-state set in which the P-states are specified as a list of one or more discrete values.

Syntax

```
typedef struct _PEP_PERF_STATE {
  ULONGLONG Value;
  PVOID     Context;
} PEP_PERF_STATE, *PPEP_PERF_STATE;
```

Members

`Value`

The discrete value represented by this P-state. For more information, see Remarks.

`Context`

A pointer to PEP-defined context data. The PEP uses this context to contain additional information about the discrete performance level that cannot be expressed in the **Value** member. This context is opaque to the Windows power management framework (PoFx). The **Context** member is optional and can be set to NULL.

Remarks

The **Discrete.States** member of the PEP_COMPONENT_PERF_SET structure is a pointer to an array of **PEP_PERF_STATE** structures. The **Unit** member of the **PEP_COMPONENT_PERF_SET** structure specifies the units in which the **Value** member in each array element is expressed. Component performance can be expressed in hertz (frequency) or in bits per second (bandwidth). For example, if **Value** = 100,000,000 and **Unit** = **PepPerfStateUnitFrequency**, this performance state represents a frequency of 100 megahertz.

Device drivers use the PO_FX_PERF_STATE structure, which is similar to the **PEP_PERF_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PO_FX_PERF_STATE

PEP_PERF_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE_TYPE** enumeration indicates the type of performance information that is specified for a performance state (P-state) of a component.

Syntax

```
typedef enum _PEP_PERF_STATE_TYPE {
  PepPerfStateTypeDiscrete,
  PepPerfStateTypeRange,
  PepPerfStateTypeMax
} PEP_PERF_STATE_TYPE, *PPEP_PERF_STATE_TYPE;
```

Constants

PepPerfStateTypeDiscrete	The performance information is specified as a set of discrete values.
PepPerfStateTypeRange	The performance information is specified as a range of values.
PepPerfStateTypeMax	Reserved for use by operating system.

Remarks

The **Type** member of the PEP_COMPONENT_PERF_SET structure is a **PEP_PERF_STATE_TYPE** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PEP_PERF_STATE_UNIT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PERF_STATE_UNIT** enumeration indicates the measurement units in which the performance state (P-state) of a component is specified.

Syntax

```
typedef enum _PEP_PERF_STATE_UNIT {
  PepPerfStateUnitOther,
  PepPerfStateUnitFrequency,
  PepPerfStateUnitBandwidth,
  PepPerfStateUnitMax
} PEP_PERF_STATE_UNIT, *PPEP_PERF_STATE_UNIT;
```

Constants

PepPerfStateUnitOther	Some other type of measurement units not explicitly described by this enumeration.
PepPerfStateUnitFrequency	Hertz (frequency units).
PepPerfStateUnitBandwidth	Bits per second (bandwidth units).
PepPerfStateUnitMax	Reserved for use by operating system.

Remarks

The **Type** member of the PEP_COMPONENT_PERF_SET structure is a **PEP_PERF_STATE_UNIT** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_SET

PEP_PLATFORM_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PLATFORM_IDLE_STATE** structure specifies the properties of a platform idle state.

Syntax

```
typedef struct _PEP_PLATFORM_IDLE_STATE {
  POHANDLE                      InitiatingProcessor;
  UCHAR                         InitiatingState;
  ULONG                         Latency;
  ULONG                         BreakEvenDuration;
  ULONG                         DependencyArrayUsed;
  ULONG                         DependencyArrayCount;
  PEP_PROCESSOR_IDLE_DEPENDENCY DependencyArray[ANYSIZE_ARRAY];
} PEP_PLATFORM_IDLE_STATE, *PPEP_PLATFORM_IDLE_STATE;
```

Members

`InitiatingProcessor`

A **POHANDLE** value that identifies the processor that initiates the transition to this platform idle state, or **NULL** if any processor can initiate the transition. If non-NULL, this handle represents the registration of the processor (as a device) with the Windows power management framework (PoFx).

`InitiatingState`

The index of the processor idle state that the processor enters to initiate the platform's entry to the specified platform idle state. If the **IdleStates** array in the PEP_PPM_QUERY_IDLE_STATES_V2 structure contains N elements, the idle states are numbered 0 to N-1 in the order in which they appear in the array.

`Latency`

The worst-case latency, in 100-nanosecond units, that the platform requires to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. PoFx uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

`DependencyArrayUsed`

The number of items in **DependencyArray** which were filled in by the PEP.

`DependencyArrayCount`

The number of elements in the **DependencyArray** array. The array contains one element for each processor in the hardware platform.

`DependencyArray`

The first element in an array of PEP_PROCESSOR_IDLE_DEPENDENCY structures. This array specifies the set of

dependencies that the platform idle state has on each processor. If the platform contains N processors, the array contains N elements, and processors are numbered 0 to N-1 in the order in which they are represented in the array.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification. The **State** member of a PEP_PPM_QUERY_PLATFORM_STATE structure is a **PEP_PLATFORM_IDLE_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PPM_QUERY_PLATFORM_STATE

PEP_PROCESSOR_IDLE_DEPENDENCY

PEP_PLATFORM_IDLE_STATE_UPDATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PLATFORM_IDLE_STATE_UPDATE** structure contains the updated properties of a platform idle state.

Syntax

```
typedef struct _PEP_PLATFORM_IDLE_STATE_UPDATE {
  ULONG Version;
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PLATFORM_IDLE_STATE_UPDATE, *PPEP_PLATFORM_IDLE_STATE_UPDATE;
```

Members

`Version`

The current version number of this structure. Set this member to PEP_PLATFORM_IDLE_STATE_UPDATE_VERSION.

`Latency`

The worst-case latency, in 100-nanosecond units, for the platform to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. The Windows power management framework (PoFx) uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

The *Update* parameter of the UpdatePlatformIdleState routine is a pointer to a **PEP_PLATFORM_IDLE_STATE_UPDATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

UpdatePlatformIdleState

PEP_POWER_CONTROL_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_POWER_CONTROL_COMPLETE** structure contains status information for a power control operation that the PEP previously requested and that the device driver has completed.

Syntax

```
typedef struct _PEP_POWER_CONTROL_COMPLETE {
  PEPHANDLE DeviceHandle;
  LPCGUID   PowerControlCode;
  PVOID     RequestContext;
  SIZE_T    BytesReturned;
  NTSTATUS  Status;
} PEP_POWER_CONTROL_COMPLETE, *PPEP_POWER_CONTROL_COMPLETE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation that was performed. This is the same value that the PEP supplied in response to the PEP_DPM_WORK notification to initiate the power control operation.

`RequestContext`

[in] A pointer to the request context that was sent by the PEP in the PEP_WORK_INFORMATION structure that the PEP supplied in the original work request. Typically, this member points to a structure that contains a pointer to an output buffer to contain the results of the power control operation that was requested by the PEP.

`BytesReturned`

[in] The size, in bytes, of the result data stored by the driver in the output buffer. For more information about this buffer, see the description of the **RequestContext** member.

`Status`

[in] The status of the power control operation. If the operation was successful, the PEP sets this member to STATUS_SUCCESS. Otherwise, the PEP sets this member to an appropriate error status code.

Remarks

This structure is used by the PEP_DPM_POWER_CONTROL_COMPLETE notification. All five members of the structure contain input values that are supplied by

If the output buffer is too small to receive all of the result data from the operation, the PEP sets the **Status** member of the structure to STATUS_INSUFFICIENT_RESOURCES, sets the **BytesReturned** member to the required size of the output buffer, and (typically) writes no data to the output buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

GUID

PEP_POWER_CONTROL_REQUEST structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_POWER_CONTROL_REQUEST** structure contains a request from a driver for a power control operation.

Syntax

```
typedef struct _PEP_POWER_CONTROL_REQUEST {
  PEPHANDLE DeviceHandle;
  LPCGUID   PowerControlCode;
  PVOID     InBuffer;
  SIZE_T    InBufferSize;
  PVOID     OutBuffer;
  SIZE_T    OutBufferSize;
  SIZE_T    BytesReturned;
  NTSTATUS  Status;
} PEP_POWER_CONTROL_REQUEST, *PPEP_POWER_CONTROL_REQUEST;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation to perform. This is the same value that the requesting driver supplied as the *PowerControlCode* parameter to the PoFxPowerControl routine.

`InBuffer`

[in] A pointer to a driver-allocated input buffer that contains the input parameters for this power control operation. This is the same value that the requesting driver supplied as the *InBuffer* parameter to the **PoFxPowerControl** routine.

`InBufferSize`

[in] The size in bytes of the buffer pointed to by **InBuffer**. This is the same value that the requesting driver supplied as the *InBufferSize* parameter to the **PoFxPowerControl** routine.

`OutBuffer`

[in] A pointer to a driver-allocated output buffer to which the PEP writes the results of this power control operation. This is the same value that the requesting driver supplied as the *OutBuffer* parameter to the **PoFxPowerControl** routine.

`OutBufferSize`

[in] The size in bytes of the buffer pointed to by **OutBuffer**.

`BytesReturned`

[out] The number of bytes that the PEP wrote to the output buffer pointed to by **OutBuffer**.

`Status`

[out] The status of the requested power control operation. If the operation is successful, the PEP sets this member to STATUS_SUCCESS. Otherwise, the PEP sets this member to an appropriate error status code.

Remarks

This structure is used by the PEP_DPM_POWER_CONTROL_REQUEST notification. The first six members of the structure contain input values that are supplied by the Windows power management Framework (PoFx). The last two members contain values that the PEP writes to the structure in response to this notification. In addition, the PEP writes to the output buffer pointed to by the **OutBuffer** member.

If the output buffer is too small to receive all of the result data from the operation, the PEP sets the **Status** member of the structure to STATUS_INSUFFICIENT_RESOURCES, sets the **BytesReturned** member to the required size of the output buffer, and (typically) writes no data to the output buffer.

The driver for a device can call the PoFxPowerControl routine to request a power control operation on the device. During this call, PoFx delegates the handling of this request to the PEP that owns the device. The **PowerControlCode** member of the **PEP_POWER_CONTROL_REQUEST** structure contains a custom control code whose meaning is understood by both the driver and the PEP for the device.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_POWER_CONTROL_REQUEST

PEP_DPM_REGISTER_DEVICE

PoFxPowerControl

PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CONTEXT_QUERY_PARKING_PAGE** structure describes the parking page for a processor.

Syntax

```
typedef struct _PEP_PPM_CONTEXT_QUERY_PARKING_PAGE {
  PHYSICAL_ADDRESS PhysicalPageAddress;
  PVOID            VirtualPageAddress;
} PEP_PPM_CONTEXT_QUERY_PARKING_PAGE, *PPEP_PPM_CONTEXT_QUERY_PARKING_PAGE;
```

Members

`PhysicalPageAddress`

The physical memory address of the parking page.

`VirtualPageAddress`

The virtual memory address of the parking page.

Remarks

The output buffer for a PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE power control request is a **PEP_PPM_CONTEXT_QUERY_PARKING_PAGE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE

PEP_PPM_CST_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CST_STATE** structure specifies the properties of a C state (ACPI processor power state).

Syntax

```
typedef struct _PEP_PPM_CST_STATE {
  UCHAR            Type;
  USHORT           Latency;
  ULONG            Power;
  UCHAR            AddressSpaceId;
  UCHAR            BitWidth;
  UCHAR            BitOffset;
  UCHAR            AccessSize;
  PHYSICAL_ADDRESS Address;
} PEP_PPM_CST_STATE, *PPEP_PPM_CST_STATE;
```

Members

`Type`

The C-state type (0 = C0, 1 = C1, 2 = C2, and so on).

`Latency`

The worst-case latency, in microseconds, to enter and exit this C state. There are no restrictions on latency size.

`Power`

The average power consumption, in milliwatts, of the processor when in this C state.

`AddressSpaceId`

The register address space ID (_ASI) for this C state. This member specifies the address space in which the register for this C state is located. The following IDs are defined for this member.

ID	MEANING
0x00	System memory space
0x01	System I/O space
0x02	PCI configuration space
0x03	Embedded controller
0x04	SMBus address
0x0A	Platform communications channel (PCC)
0x7F	Functional fixed hardware

`BitWidth`

The register bit width (_RBW) for this C state. This member specifies the width, in bits, of the register for this C state.

`BitOffset`

The register bit offset (_RBO) for this C state. This member specifies the offset, in bits, from the register address specified in the **Address** member to the start of the register for this C state.

`AccessSize`

The register access size (_ASZ) code for this C state. This member specifies the number of bytes to read during an access of the register for this C state. Valid access size codes are shown in the following table.

ACCESS SIZE CODE	MEANING
0	Undefined
1	BYTE (1 byte)
2	WORD (2 bytes)
3	DWORD (4 bytes)
4	QWORD (8 bytes)

`Address`

The register address (_ADR) for this C state.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_CST_STATES notification. The **IdleStates** member of the PEP_PPM_CST_STATES structure is a pointer to an array of **PEP_PPM_CST_STATE** structures. The **Type** member in each element of the array should equal the array index of this member.

The values in the **PEP_PPM_CST_STATE** structure are obtained from the parameters that are specified for this C state in the _CST object in the ACPI namespace for the processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_CST_STATES

PEP_PPM_CST_STATES

PEP_PPM_CST_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_CST_STATES** structure specifies the properties of the C states (ACPI processor power states) that are supported for a processor.

Syntax

```
typedef struct _PEP_PPM_CST_STATES {
  ULONG              Count;
  PPEP_PPM_CST_STATE IdleStates;
} PEP_PPM_CST_STATES, *PPEP_PPM_CST_STATES;
```

Members

`Count`

The number of elements in the **IdleStates** array.

`IdleStates`

A pointer to an array of PEP_PPM_CST_STATE structures.

Remarks

This structure is used by the PEP_NOTIFY_PPM_CST_STATES notification. The contents of this structure are obtained from the _CST object that is located in the ACPI namespace for the processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_CST_STATES

PEP_PPM_CST_STATE

PEP_PPM_ENTER_SYSTEM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE** notification to notify PEP that the system is about to enter a system power state.

Syntax

```
typedef struct _PEP_PPM_ENTER_SYSTEM_STATE {
  SYSTEM_POWER_STATE TargetState;
} PEP_PPM_ENTER_SYSTEM_STATE, *PPEP_PPM_ENTER_SYSTEM_STATE;
```

Members

`TargetState`

On input, a SYSTEM_POWER_STATE-type value that indicates the power state that the system is entering.

Remarks

This notification is sent to all processors simultaneously after the system has completed all passive level work transitioning the processor to the system power state. This notification is sent at DISPATCH_LEVEL, with all processors at dispatch. This notification is always executed on the target processor. PEP must not queue any work from this notification. The processors will not process work items, DPCs, etc. after this notification has been sent.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pepfx.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_FEEDBACK_READ structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_FEEDBACK_READ** structure contains the value read from a processor performance feedback counter.

Syntax

```
typedef struct _PEP_PPM_FEEDBACK_READ {
  ULONG CounterIndex;
  union {
    ULONG64 InstantaneousValue;
    struct {
      ULONG64 NominalCount;
      ULONG64 ActualCount;
    };
  };
} PEP_PPM_FEEDBACK_READ, *PPEP_PPM_FEEDBACK_READ;
```

Members

`CounterIndex`

[in] The index that identifies which processor performance feedback counter to read. If the platform extension plug-in (PEP) supports N counters for this processor, counter indexes range from 0 to N-1. The PEP previously supplied the number of supported counters in response to a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification.

`InstantaneousValue`

`NominalCount`

`ActualCount`

Remarks

This structure is used by the PEP_NOTIFY_PPM_FEEDBACK_READ notification. The **CounterIndex** member of the structure contains an input value supplied by the Windows power management framework (PoFx) when this notification is set. The other members contain output values that the PEP writes to the structure in response to the notification. The PEP writes to the **InstantaneousValue** member if the counter generates an instantaneous value, or to the **NominalCount** and **ActualCount** members if the counter generates a relative value.

Both an instantaneous counter and a relative counter are reset to zero when power is first turned on, but reading a relative counter causes the count to reset to zero, whereas reading an instantaneous counter does not reset the count. The PEP previously indicated whether the counter is instantaneous or relative in response to a PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_FEEDBACK_READ

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

PEP_PPM_IDLE_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_COMPLETE** structure describe the idle states from which the processor and hardware platform are waking.

Syntax

```
typedef struct _PEP_PPM_IDLE_COMPLETE {
  ULONG ProcessorState;
  ULONG PlatformState;
} PEP_PPM_IDLE_COMPLETE, *PPEP_PPM_IDLE_COMPLETE;
```

Members

`ProcessorState`

[in] An index that identifies the idle state from which the processor is waking. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1. When the hypervisor is enabled and the platform wakes from an idle state, this member is set to **PEP_PROCESSOR_IDLE_STATE_UNKNOWN** (0xffffffff). For more information, see PEP_NOTIFY_PPM_IDLE_COMPLETE.

`PlatformState`

[in] An index that identifies the idle state from which the platform is waking. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If the transition from the processor idle state does not involve a transition from an platform idle state, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_COMPLETE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_COMPLETE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_IDLE_COMPLETE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_COMPLETE_V2** structure describe the idle states from which the processor and hardware platform are waking.

Syntax

```
typedef struct _PEP_PPM_IDLE_COMPLETE_V2 {
  ULONG  ProcessorState;
  ULONG  PlatformState;
  ULONG  CoordinatedStateCount;
  PULONG CoordinatedStates;
} PEP_PPM_IDLE_COMPLETE_V2, *PPEP_PPM_IDLE_COMPLETE_V2;
```

Members

`ProcessorState`

[in] An index that identifies the idle state from which the processor is waking. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1. When the hypervisor is enabled and the platform wakes from an idle state, this member is set to **PEP_PROCESSOR_IDLE_STATE_UNKNOWN** (0xffffffff). For more information, see PEP_NOTIFY_PPM_IDLE_COMPLETE.

`PlatformState`

[in] An index that identifies the idle state from which the platform is waking. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If the transition from the processor idle state does not involve a transition from an platform idle state, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`CoordinatedStateCount`

`CoordinatedStates`

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_COMPLETE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

PEP_PPM_IDLE_EXECUTE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_EXECUTE** structure specifies the idle state that the processor is to enter.

Syntax

```
typedef struct _PEP_PPM_IDLE_EXECUTE {
  NTSTATUS Status;
  ULONG    ProcessorState;
  ULONG    PlatformState;
} PEP_PPM_IDLE_EXECUTE, *PPEP_PPM_IDLE_EXECUTE;
```

Members

`Status`

[out] An **NTSTATUS** value that indicates whether the processor idle state transition was successful. The platform extension plug-in (PEP) sets this member to **STATUS_SUCCESSFUL** if the transition succeeded. Otherwise, this member is set to an appropriate error status code.

`ProcessorState`

[in] The index of the processor idle state that the processor is to enter. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value P**EP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_EXECUTE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The **Status** member contains an output value that the PEP writes to this member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_EXECUTE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_IDLE_EXECUTE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IDLE_EXECUTE_V2** structure specifies the idle state that the processor is to enter.

Syntax

```
typedef struct _PEP_PPM_IDLE_EXECUTE_V2 {
  NTSTATUS Status;
  ULONG    ProcessorState;
  ULONG    PlatformState;
  ULONG    CoordinatedStateCount;
  PULONG   CoordinatedStates;
} PEP_PPM_IDLE_EXECUTE_V2, *PPEP_PPM_IDLE_EXECUTE_V2;
```

Members

`Status`

[out] An **NTSTATUS** value that indicates whether the processor idle state transition was successful. The platform extension plug-in (PEP) sets this member to **STATUS_SUCCESSFUL** if the transition succeeded. Otherwise, this member is set to an appropriate error status code.

`ProcessorState`

[in] The index of the processor idle state that the processor is to enter. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`CoordinatedStateCount`

Supplies the number of coordinated idle states being entered by this transition.

`CoordinatedStates`

Supplies a pointer to an array of coordinated idle states that are being entered by this transition.

Remarks

This structure is used by the PEP_NOTIFY_PPM_IDLE_EXECUTE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The **Status** member contains an output value that the PEP writes to this member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IDLE_EXECUTE

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_INITIATE_WAKE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_INITIATE_WAKE** structure indicates whether a processor requires an interrupt to wake up from an idle state.

Syntax

```
typedef struct _PEP_PPM_INITIATE_WAKE {
  BOOLEAN NeedInterruptForCompletion;
} PEP_PPM_INITIATE_WAKE, *PPEP_PPM_INITIATE_WAKE;
```

Members

`NeedInterruptForCompletion`

[out] Whether the processor requires an interrupt finish waking up from an idle state. Set to **TRUE** if the processor requires an interrupt, or to **FALSE** if it does not.

Remarks

This structure is used by the PEP_NOTIFY_PPM_INITIATE_WAKE notification. The **NeedInterruptForCompletion** member contains an output value that the platform extension plug-in (PEP) writes to the structure in response to this notification.

If the PEP sets the **NeedInterruptForCompletion** member to TRUE, the PEP must ensure that the processor is enabled to be interrupted before the AcceptProcessorNotification callback routine returns.

The PEP should set the **NeedInterruptForCompletion** member to **FALSE** if the processor is already running and/or will eventually exit the idle state (and is in the process of doing so) without requiring a software-generated interrupt.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

AcceptProcessorNotification

PEP_NOTIFY_PPM_INITIATE_WAKE

PEP_PPM_IS_PROCESSOR_HALTED structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_IS_PROCESSOR_HALTED** structure indicates whether the processor is currently halted in its selected idle state.

Syntax

```
typedef struct _PEP_PPM_IS_PROCESSOR_HALTED {
  BOOLEAN Halted;
} PEP_PPM_IS_PROCESSOR_HALTED, *PPEP_PPM_IS_PROCESSOR_HALTED;
```

Members

`Halted`

[out] Whether the processor is halted. Set to **TRUE** if the processor is halted, or to **FALSE** if it is not halted.

Remarks

This structure is used by the PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED notification. The **Halted** member contains an output value that the platform extension plug-in (PEP) writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED

PEP_PPM_PARK_MASK structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_MASK** structure contains the current core parking mask.

Syntax

```
typedef struct _PEP_PPM_PARK_MASK {
  ULONG                     Count;
  ULONGLONG                 EvaluationTime;
  PPEP_PROCESSOR_PARK_STATE Processors;
} PEP_PPM_PARK_MASK, *PPEP_PPM_PARK_MASK;
```

Members

`Count`

[in] Indicates the number of processors in the **Processors** array.

`EvaluationTime`

[in] The interrupt time of the performance check evaluation that initiated this notification.

`Processors`

[in/out] An array of processors in the core parking domain.

On input

Indicates the OS parking preference for each processor.

On output

Returns the PEP parking preference for each processor.

The Processors array is guaranteed to contain the processors in increasing order by processor index.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_MASK notification

PEP_PROCESSOR_PARK_STATE

PEP_PPM_PARK_SELECTION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PARK_SELECTION** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PPM_PARK_SELECTION {
  ULONG                          AdditionalUnparkedProcessors;
  ULONG                          Count;
  PPEP_PROCESSOR_PARK_PREFERENCE Processors;
} PEP_PPM_PARK_SELECTION, *PPEP_PPM_PARK_SELECTION;
```

Members

`AdditionalUnparkedProcessors`

[in] The number of additional processors that need to be parked.

This number includes all processors in the **Processors** array for which the PEP sets **PepPreference** to PROCESSOR_PARK_PREFERENCE_PARKED but the operating system sets **PoPreference** to either PROCESSOR_PARK_PREFERENCE_UNPARKED or PROCESSOR_PARK_PREFERENCE_NONE. For more information about these values, see PEP_PROCESSOR_PARK_PREFERENCE.

This number does not include processors in the **Processors** array for which the operating system sets the **PoPreference** value to PROCESSOR_PARK_PREFERENCE_PARKED.

`Count`

[in] The number of elements in the array pointed to by the **Processors** member.

`Processors`

[in/out] A pointer to an array of PEP_PROCESSOR_PARK_PREFERENCE structures. The **Count** member specifies the number of elements in the array.

On input

Each array element indicates the parking preferences for a particular processor.

On output

Returns the PEP parking preference for each processor.

The Processors array is guaranteed to contain the processors in increasing order by processor index.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PARK_SELECTION notification. All three members of the structure contain input values that are set by the Windows power management framework (PoFx) before this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_SELECTION

PEP_PROCESSOR_PARK_PREFERENCE

PEP_PPM_PARK_SELECTION_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PARK_SELECTION_V2** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding which processors in the platform should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PPM_PARK_SELECTION_V2 {
  ULONG                          AdditionalUnparkedProcessors;
  ULONG                          Count;
  PPEP_PROCESSOR_PARK_PREFERENCE Processors;
  ULONGLONG                      EvaluationTime;
  UCHAR                          EvaluationType;
} PEP_PPM_PARK_SELECTION_V2, *PPEP_PPM_PARK_SELECTION_V2;
```

Members

`AdditionalUnparkedProcessors`

[in] The number of additional processors that need to be unparked.

This number includes all processors in the **Processors** array for which the PEP sets **PepPreference** to PROCESSOR_PARK_PREFERENCE_PARKED but the operating system sets **PoPreference** to either PROCESSOR_PARK_PREFERENCE_UNPARKED or PROCESSOR_PARK_PREFERENCE_NONE. For more information about these values, see PEP_PROCESSOR_PARK_PREFERENCE.

This number does not include processors in the **Processors** array for which the operating system sets the **PoPreference** value to PROCESSOR_PARK_PREFERENCE_PARKED.

`Count`

[in] The number of elements in the array pointed to by the **Processors** member.

`Processors`

[in/out] A pointer to an array of PEP_PROCESSOR_PARK_PREFERENCE structures. Each array element indicates the parking preferences for a particular processor. The **Count** member specifies the number of elements in the array.

`EvaluationTime`

The interrupt time of the performance check evaluation that initiated this notification.

`EvaluationType`

The type of evaluation being performed. This can be one of the following values:

VALUE	MEANING

PROCESSOR_PARKING_TYPE_CORE_PARKING 0x00	The processor should be parked.
PROCESSOR_PARKING_TYPE_INTERRUPT_STEERING 0x01	Interrupts should be steered away from the processor.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PARK_SELECTION_V2 notification notification. All three members of the structure contain input values that are set by the Windows power management framework (PoFx) before this notification is sent to the PEP.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

PEP_PPM_PERF_CHECK_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_CHECK_COMPLETE** structure is used to inform the PEP of details regarding the completion of a periodic performance check evaluation.

Syntax

```
typedef struct _PEP_PPM_PERF_CHECK_COMPLETE {
  ULONGLONG EvaluationTime;
} PEP_PPM_PERF_CHECK_COMPLETE, *PPEP_PPM_PERF_CHECK_COMPLETE;
```

Members

`EvaluationTime`

[in] The interrupt time of the performance check evaluation that initiated this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_CHECK_COMPLETE notification

PEP_PPM_PERF_SET structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_SET** structure specifies the new performance level that the operating system is requesting for the processor.

Syntax

```
typedef struct _PEP_PPM_PERF_SET {
  ULONG MinimumPerformance;
  ULONG MaximumPerformance;
  ULONG DesiredPerformance;
  ULONG TimeWindow;
  ULONG PerformanceTolerance;
} PEP_PPM_PERF_SET, *PPEP_PPM_PERF_SET;
```

Members

`MinimumPerformance`

[in] The new minimum performance level in platform-specific units. This member indicates the absolute minimum performance level that the processor can run at.

`MaximumPerformance`

[in] The new maximum performance level in platform-specific units. This member indicates the absolute maximum performance level that the processor can run at.

`DesiredPerformance`

[in] The new desired performance level in platform-specific units. This member is set to a value in the range **MinimumPerformance** to **MaximumPerformance**.

`TimeWindow`

[in] The width, in milliseconds, of the new time window over which the platform must provide the required average performance. If this time window is nonzero, the instantaneous performance provided by the platform does not need to match the desired performance level, and only needs to fall within the constraints specified by the minimum and maximum performance level.

`PerformanceTolerance`

[in] The new performance tolerance in platform-specific units. This member contains the minimum performance level that the platform can deliver and still meet the operating system's performance requirements.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PERF_SET notification. All five members of this structure contain input values that PoFx supplies when this notification is sent to the PEP.

Processor performance levels are specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done. For more information, see Platform Performance Thresholds.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_SET

Platform Performance Thresholds

PEP_PPM_PERF_SET_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_PERF_SET** notification at runtime to set the current operating performance of the processor.

Syntax

```
typedef struct _PEP_PPM_PERF_SET_STATE {
  UCHAR MinimumPerformanceState;
  UCHAR MaximumPerformanceState;
  UCHAR DesiredPerformanceState;
} PEP_PPM_PERF_SET_STATE, *PPEP_PPM_PERF_SET_STATE;
```

Members

`MinimumPerformanceState`

On input, the new requested minimum performance state. This state is defined as the absolute minimum instantaneous performance level at which the processor may run. The operating system may set this value to any performance value in the range [Lowest Performance, Guaranteed Performance], inclusive. Minimum performance must never be set to a value higher than maximum performance.

`MaximumPerformanceState`

On input, the new requested maximum performance state. This state is defined as the absolute maximum instantaneous performance level at which the processor may run. The OS may set Maximum performance to any performance value in the range [Lowest Performance, Highest Performance], inclusive.

`DesiredPerformanceState`

On input, the new requested desired performance state. This state is defined as the performance level power manager is requesting from the processor. Desired performance may be set to any performance value in the range [Minimum Performance, Maximum Performance], inclusive. Desired performance may take one of two meanings, depending on whether the desired performance is above or below the guaranteed performance level.

- Below the guaranteed performance level, desired performance expresses the average performance level the platform must provide.
- Above the guaranteed performance level, the platform must provide the guaranteed performance level. The platform should attempt to provide up to the desired performance level, if current operating conditions allow for it, but it is not required to do so.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Header	pepfx.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_PLATFORM_STATE_RESIDENCIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PLATFORM_STATE_RESIDENCIES** structure contains the accumulated residency times and transition counts for the idle states that are supported by the hardware platform.

Syntax

```
typedef struct _PEP_PPM_PLATFORM_STATE_RESIDENCIES {
  ULONG                              Count;
  PPEP_PPM_PLATFORM_STATE_RESIDENCY States;
} PEP_PPM_PLATFORM_STATE_RESIDENCIES, *PPEP_PPM_PLATFORM_STATE_RESIDENCIES;
```

Members

`Count`

[in] The number of elements in the array pointed to by the **States** member. The number of array elements equals the number of platform idle states that the PEP previously provided in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

`States`

[in] A pointer to an output buffer to which the PEP writes an array of PEP_PPM_PLATFORM_STATE_RESIDENCY structures. Each structure contains the accumulated residency and transition count information for a particular platform idle state. This buffer is guaranteed to be large enough to contain the number of array elements specified by the **Count** member. Each idle state is identified by its array index. If the array contains N elements, idle state indexes range from 0 to N-1.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES notification. Both members of the structure contain input values that the Windows power management framework (PoFx) supplies when this notification is sent to the PEP. PoFx allocates the output buffer pointed to by the **States** member, and the PEP writes to this buffer.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES

PEP_PPM_PLATFORM_STATE_RESIDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PLATFORM_STATE_RESIDENCY** structure specifies the accumulated residency time and transition count for a particular platform idle state.

Syntax

```
typedef struct _PEP_PPM_PLATFORM_STATE_RESIDENCY {
  ULONG64 Residency;
  ULONG64 TransitionCount;
} PEP_PPM_PLATFORM_STATE_RESIDENCY, *PPEP_PPM_PLATFORM_STATE_RESIDENCY;
```

Members

`Residency`

The accumulated residency time, in 100-nanosecond units. This is the total amount of time that the hardware platform has spent in this idle state since system startup.

`TransitionCount`

The total number of times that the platform has entered this idle state since system startup.

Remarks

The **States** member of the PEP_PPM_PLATFORM_STATE_RESIDENCIES structure is a pointer to an array of **PEP_PPM_PLATFORM_STATE_RESIDENCY** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_PPM_PLATFORM_STATE_RESIDENCIES

PEP_PPM_QUERY_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_CAPABILITIES** structure contains information about the processor power management (PPM) capabilities of the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_PPM_QUERY_CAPABILITIES {
  ULONG   FeedbackCounterCount;
  ULONG   IdleStateCount;
  BOOLEAN PerformanceStatesSupported;
  BOOLEAN ParkingSupported;
  UCHAR   DiscretePerformanceStateCount;
} PEP_PPM_QUERY_CAPABILITIES, *PPEP_PPM_QUERY_CAPABILITIES;
```

Members

`FeedbackCounterCount`

[out] The number of processor performance feedback counters supported by the PEP for this processor. On x86/AMD64 platforms, this must be zero.

`IdleStateCount`

[out] The number of processor idle states that the PEP supports. The PEP is not required to support the same number of idle states for all processors. The PEP can set **IdleStateCount** = 0 to indicate that it doesn't support any idle states, in which case the PEP doesn't need to implement any other notifications that deal with processor idle states.

`PerformanceStatesSupported`

[out] Whether the PEP supports performance state controls. Set to TRUE if the PEP supports performance states, or to FALSE if it does not.

`ParkingSupported`

[out] Whether the PEP supports supplying core parking hints. Set to TRUE if the PEP can indicate which cores to park, or to FALSE if it cannot.

`DiscretePerformanceStateCount`

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification. All four members contain output values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_COORDINATED_DEPENDENCY** structure describes dependencies for coordinated idle states.

Syntax

```
typedef struct _PEP_PPM_QUERY_COORDINATED_DEPENDENCY {
  ULONG                              StateIndex;
  ULONG                              DependencyIndex;
  ULONG                              DependencySize;
  ULONG                              DependencySizeUsed;
  POHANDLE                           TargetProcessor;
  PEP_COORDINATED_DEPENDENCY_OPTION Options[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_COORDINATED_DEPENDENCY, *PPEP_PPM_QUERY_COORDINATED_DEPENDENCY;
```

Members

`StateIndex`

[in] The index of the coordinated idle state which is having its dependencies queried.

`DependencyIndex`

[in] The index of the dependency being queried.

`DependencySize`

[in] The size of the **Dependencies** array.

`DependencySizeUsed`

[out] The number of elements of the **Dependencies** array filled in by the PEP.

`TargetProcessor`

[out] The **POHANDLE** corresponding to the processor being targeted by this dependency, or **NULL** if this is a coordinated state dependency.

`Options`

[out] A list of PEP_COORDINATED_DEPENDENCY_OPTION structures describing dependency options, one of which must be satisfied for this coordinated state to be entered.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

See Also

PEP_COORDINATED_DEPENDENCY_OPTION

PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY notification

PEP_PPM_QUERY_COORDINATED_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_COORDINATED_STATES** structure contains information about each coordinated idle state that the platform extension plug-in (PEP) supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_COORDINATED_STATES {
  ULONG                      Count;
  PEP_COORDINATED_IDLE_STATE States[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_COORDINATED_STATES, *PPEP_PPM_QUERY_COORDINATED_STATES;
```

Members

`Count`

[in] The size of the **States** array. This will be equal to the **PlatformStateCount** returned from the query platform states notification.

`States`

[out] An array of PEP_COORDINATED_IDLE_STATE structures that contains information about each coordinated idle state that the PEP supports.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COORDINATED_IDLE_STATE

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification

PEP_PPM_QUERY_DISCRETE_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES** notification that stores the list of discrete performance states that PEP supports, if the **PEP_NOTIFY_PPM_QUERY_CAPABILITIES** notification indicates support for discrete performance states.

Syntax

```
typedef struct _PEP_PPM_QUERY_DISCRETE_PERF_STATES {
  ULONG                     Count;
  PPEP_PROCESSOR_PERF_STATE States;
} PEP_PPM_QUERY_DISCRETE_PERF_STATES, *PPEP_PPM_QUERY_DISCRETE_PERF_STATES;
```

Members

`Count`

On input, the size of the array pointed to by *States*.

`States`

On output, an array of PEP_PROCESSOR_PERF_STATE structures that indicates performance states that is filled by PEP.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pepfx.h (include Pep_x.h)

See Also

PEP_PPM_QUERY_CAPABILITIES

Processor power management (PPM) notifications

PEP_PPM_QUERY_DOMAIN_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

Used in the **PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO** notification that queries for information about a performance domain.

Syntax

```
typedef struct _PEP_PPM_QUERY_DOMAIN_INFO {
  ULONG   DomainId;
  UCHAR   CoordinationType;
  BOOLEAN IdleProcessorsDiscounted;
  BOOLEAN SchedulerDirectedTransitionsSupported;
  BOOLEAN AffinitizePerfSet;
  ULONG   WorstCaseTransitionLatency;
  ULONG   WorstCaseTransitionOverhead;
} PEP_PPM_QUERY_DOMAIN_INFO, *PPEP_PPM_QUERY_DOMAIN_INFO;
```

Members

`DomainId`

On input, the domain ID of the domain being queried.

`CoordinationType`

On output, indicates the coordination type of the performance domain. The possible values are:

- PROCESSOR_DOMAIN_COORDIANTION_SW_ALL (0x00)
- PROCESSOR_DOMAIN_COORDIANTION_SW_ANY (0x01)
- PROCESSOR_DOMAIN_COORDIANTION_HW_ALL (0x02)

If this notification is not supported, PROCESSOR_DOMAIN_COORDIANTION_SW_ALL coordination is used.

`IdleProcessorsDiscounted`

On output, and if HW-ALL coordination is used, this value is TRUE if the platform automatically ignores requests from idle processors as it determines the resolved performance level of the domain, FALSE if requests from idle processors continue to be honored.
If PROCESSOR_DOMAIN_COORDIANTION_HW_ALL coordination is not used: this field is ignored.

`SchedulerDirectedTransitionsSupported`

On output, this value is TRUE if the PEP supports the requirements of scheduler directed performance transitions for this domain, FALSE otherwise.

`AffinitizePerfSet`

On output, this value is TRUE if the PEP requires the PEP_NOTIFY_PPM_PERF_SET or PEP_NOTIFY_PPM_PERF_SET_STATE notification to execute on the processor changing frequency, or FALSE if the PEP allows these notifications to execute on any processor. If PROCESSOR_DOMAIN_COORDIANTION_SW_ANY coordination is used: the PEP_NOTIFY_PPM_PERF_SET or PEP_NOTIFY_PPM_PERF_SET_STATE notification will execute on a processor in the domain.

`WorstCaseTransitionLatency`

On output, this value indicates the worst case latency for the processor to transition to a new performance state, measured from the time the OS issues the notification to the PEP to transition to a new performance level, to the time the processor is running at the new performance level, in 100ns units.

`WorstCaseTransitionOverhead`

On output, this value indicates the worst case overhead (PEP code execution time plus time the processor is not executing instructions) measured from the time the OS issues the notification to the PEP to transition to a new performance level, to the time the processor is running at the new performance level, in 100ns units.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pepfx.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_QUERY_FEEDBACK_COUNTERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure describes all the processor performance counters that the platform extension plug-in (PEP) supports for a particular processor.

Syntax

```
typedef struct _PEP_PPM_QUERY_FEEDBACK_COUNTERS {
  ULONG                          Count;
  PEP_PROCESSOR_FEEDBACK_COUNTER Counters[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_FEEDBACK_COUNTERS, *PPEP_PPM_QUERY_FEEDBACK_COUNTERS;
```

Members

`Count`

[in] The number of processor performance feedback counters supported by the PEP. The PEP previously supplied this count in response to a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification.

`Counters`

[out] The first element in an array of PEP_PROCESSOR_FEEDBACK_COUNTER structures. If this array contains more than one element, the additional elements immediately follow the end of the **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure. The **Count** member specifies the number of elements in this array. For more information, see Remarks.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification. The **Count** member of the structure contains an input value that PoFx supplies when this notification is sent. The **Counters** member contains an output value that the PEP writes in response to the notification. The PEP writes any additional **Counters** array elements to the output buffer area that follows the **PEP_PPM_QUERY_FEEDBACK_COUNTERS** structure. The buffer that PoFx allocated for this structure is guaranteed to be large enough to contain any array elements that follow the structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h, Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

PEP_PROCESSOR_FEEDBACK_COUNTER

PEP_PPM_QUERY_IDLE_STATES_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_IDLE_STATES_V2** structure is used during processor initialization to query the platform extension plug-in (PEP) for a list of processor idle states that the processor supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_IDLE_STATES_V2 {
  ULONG                       Count;
  PEP_PROCESSOR_IDLE_STATE_V2 IdleStates[ANYSIZE_ARRAY];
} PEP_PPM_QUERY_IDLE_STATES_V2, *PPEP_PPM_QUERY_IDLE_STATES_V2;
```

Members

`Count`

[in] The number of elements in the **IdleStates** array. The Windows power management framework (PoFx) previously sent the PEP a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification to determine this number.

`IdleStates`

[out] The first element in an array of PEP_PROCESSOR_IDLE_STATE_V2 structures that describe the processor idle states. If this array contains more than one element, the addition array elements follow the end of the **PEP_PPM_QUERY_IDLE_STATES_V2** structure.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. The **Count** member of the **PEP_PPM_QUERY_IDLE_STATES_V2** structure contains an input value that is supplied by the Windows power management framework (PoFx). The **IdleStates** array elements contain output values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_IDLE_STATE_V2

PEP_PPM_QUERY_PERF_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PERF_CAPABILITIES** structure describes the performance capabilities of the processors in the specified processor performance domain.

Syntax

```
typedef struct _PEP_PPM_QUERY_PERF_CAPABILITIES {
  ULONG HighestPerformance;
  ULONG NominalPerformance;
  ULONG LowestNonlinearPerformance;
  ULONG LowestPerformance;
  ULONG DomainId;
  ULONG DomainMembers;
} PEP_PPM_QUERY_PERF_CAPABILITIES, *PPEP_PPM_QUERY_PERF_CAPABILITIES;
```

Members

`HighestPerformance`

[out] The highest performance level in platform-specific units. For more information, see Remarks.

`NominalPerformance`

[out] The nominal performance level in platform-specific units. For more information, see Remarks.

`LowestNonlinearPerformance`

[out] The lowest nonlinear performance level in platform-specific units. For more information, see Remarks.

`LowestPerformance`

[out] The lowest performance level in platform-specific units. For more information, see Remarks.

`DomainId`

[out] The domain ID of the processor performance domain.

`DomainMembers`

[out] The number of processors in this performance domain.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification. All six members contain output values that the platform extension plug-in (PEP) writes to the structure in response to this notification.

Processor performance levels are specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done. For more information, see Platform Performance Thresholds.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES

Platform Performance Thresholds

PEP_PPM_PERF_CONSTRAINTS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_PERF_CONSTRAINTS** structure describes the performance limits to apply to the processor.

Syntax

```
typedef struct _PEP_PPM_QUERY_PERF_CONSTRAINTS {
  ULONG GuaranteedPerformanceLimit;
  ULONG LimitReasons;
} PEP_PPM_PERF_CONSTRAINTS, *PPEP_PPM_PERF_CONSTRAINTS;
```

Members

`GuaranteedPerformanceLimit`

[out] The guaranteed performance threshold in platform-specific units. If nonzero, this value specifies the maximum sustained performance level of the processor, taking into account all known external constraints (power budgeting, thermal constraints, power source, and so on). All processors are expected to be able to simultaneously sustain their guaranteed performance levels. For more information, see Platform Performance Thresholds.

`LimitReasons`

[out] The reasons for which the processor's performance can be limited. Set this member to zero or to the bitwise-OR of one or more of the following values:

LIMIT REASON	MEANING
PERFORMANCE_LIMIT_THERMAL 0x1	The processor performance can be limited due to an out-of-band (that is, non-ACPI Thermal Zone) thermal condition.
PERFORMANCE_LIMIT_POWER 0x2	The processor performance can be limited to keep power consumption of the processor within hardware safety limits.

PER FOR MA NCE _LI MIT _DO MAI N_D EPE NDE NC Y 0x4	The processor performance can be limited because of a voltage or frequency dependency between this processor and another, non-processor device.

Remarks

This structure is used by the PEP_NOTIFY_PPM_PERF_CONSTRAINTS notification. The **GuaranteedPerformanceLimit** and **LimitReasons** members contain output values that the platform extension plug-in (PEP) writes to the structure in response to this notification.

The **GuaranteedPerformanceLimit** member value is specified in platform-specific units. For example, a hardware platform might use a metric such as the processor clock frequency to provide a rough approximation to the amount of processing work that is being done.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PERF_CONSTRAINTS

PEP_PPM_QUERY_PLATFORM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PLATFORM_STATE** structure contains information about a platform idle state.

Syntax

```
typedef struct _PEP_PPM_QUERY_PLATFORM_STATE {
  ULONG                  StateIndex;
  PEP_PLATFORM_IDLE_STATE State;
} PEP_PPM_QUERY_PLATFORM_STATE, *PPEP_PPM_QUERY_PLATFORM_STATE;
```

Members

`StateIndex`

[in] The index of this platform idle state. If the hardware platform supports N platform idle states, the states are numbered 0 to N-1. The Windows power management framework (PoFx) previously sent a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification to the platform extension plug-in (PEP) to determine the number of supported platform idle states.

`State`

[out] A PEP_PLATFORM_IDLE_STATE structure that describes the platform idle state.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification. The **StateIndex** member of the **PEP_PPM_QUERY_PLATFORM_STATE** structure contains an input value that is supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **State** member contains an output value that the PEP writes to the structure in response to the notification.

The buffer that PoFx allocates to hold the **PEP_PPM_QUERY_PLATFORM_STATE** structure is guaranteed to be large enough to contain this structure plus any elements of the **State.DependencyArray** array that follow this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PLATFORM_IDLE_STATE

PEP_PPM_QUERY_PLATFORM_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_PLATFORM_STATES** structure specifies the number of platform idle states the hardware platform supports.

Syntax

```
typedef struct _PEP_PPM_QUERY_PLATFORM_STATES {
  ULONG PlatformStateCount;
} PEP_PPM_QUERY_PLATFORM_STATES, *PPEP_PPM_QUERY_PLATFORM_STATES;
```

Members

`PlatformStateCount`

The number of platform idle states supported by the platform.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PPM_QUERY_STATE_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_STATE_NAME** structure contains information about a specific coordinated or platform idle state.

Syntax

```
typedef struct _PEP_PPM_QUERY_STATE_NAME {
  ULONG  StateIndex;
  USHORT NameSize;
  PWSTR  Name;
} PEP_PPM_QUERY_STATE_NAME, *PPEP_PPM_QUERY_STATE_NAME;
```

Members

`StateIndex`

[in] The index of the state that the OS is querying about.

`NameSize`

[in/out] If **Name** is **NULL**, returns the size of the buffer required to hold **Name** (including the trailing **UNICODE_NULL** character), in WCHARs. If **Name** is not **NULL**, supplies the size of the **Name** buffer, in WCHARs.

`Name`

[in] A pointer to a buffer to copy the name of the state to, or **NULL**.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATE_NAME notification

PEP_NOTIFY_PPM_QUERY_PROCESSOR_STATE_NAME notification

PEP_PPM_QUERY_VETO_REASON structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_VETO_REASON** structure supplies a wide-character, null-terminated string that contains a descriptive, human-readable name for a veto reason.

Syntax

```
typedef struct _PEP_PPM_QUERY_VETO_REASON {
  ULONG  VetoReason;
  USHORT NameSize;
  PWSTR  Name;
} PEP_PPM_QUERY_VETO_REASON, *PPEP_PPM_QUERY_VETO_REASON;
```

Members

`VetoReason`

[in] The index that identifies this veto reason. If the platform extension plug-in (PEP) supports N veto reasons, valid veto reason indexes range from 1 to N. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification.

`NameSize`

[out] The size, in bytes, of the string buffer pointed to by the **Name** member. This string size includes the storage space for the null character that terminates the wide-character string.

`Name`

[in] A pointer to a PoFx-allocated string buffer. If this buffer pointer is non-**NULL**, the PEP writes a wide-character, null-terminated string to the buffer (and writes nothing to the **NameSize** member). Otherwise, the PEP writes nothing to this buffer and instead writes the required string size to the **NameSize** member.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_VETO_REASON notification. The **VetoReason** and **Name** members contain input values that are supplied by the Windows power management framework (PoFx). If the input value of the **Name** member is **NULL**, the PEP writes an output value (the required string buffer size) to the **NameSize** member.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_VETO_REASON

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_PPM_QUERY_VETO_REASONS structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_QUERY_VETO_REASONS** structure specifies the total number of veto reasons that the PEP uses in calls to the ProcessorIdleVeto and PlatformIdleVeto routines.

Syntax

```
typedef struct _PEP_PPM_QUERY_VETO_REASONS {
  ULONG VetoReasonCount;
} PEP_PPM_QUERY_VETO_REASONS, *PPEP_PPM_QUERY_VETO_REASONS;
```

Members

`VetoReasonCount`

[out] The number of veto codes used by the PEP.

Remarks

This structure is used by the PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. The **VetoReasonCount** member contains an output value that the PEP writes to this member in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_PPM_RESUME_FROM_SYSTEM_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Used by the **PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE** notification that notifies the PEP that the system has just resumed from a system power state.

Syntax

```
typedef struct _PEP_PPM_RESUME_FROM_SYSTEM_STATE {
  SYSTEM_POWER_STATE TargetState;
} PEP_PPM_RESUME_FROM_SYSTEM_STATE, *PPEP_PPM_RESUME_FROM_SYSTEM_STATE;
```

Members

`TargetState`

On input, a SYSTEM_POWER_STATE-type value that indicates the power state that the system is entering.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pepfx.h (include Pep_x.h)

See Also

Processor power management (PPM) notifications

PEP_PPM_TEST_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PPM_TEST_IDLE_STATE** structure contains information about whether the processor can immediately enter a processor idle state.

Syntax

```
typedef struct _PEP_PPM_TEST_IDLE_STATE {
  ULONG ProcessorState;
  ULONG PlatformState;
  ULONG VetoReason;
} PEP_PPM_TEST_IDLE_STATE, *PPEP_PPM_TEST_IDLE_STATE;
```

Members

`ProcessorState`

(input) The index of the processor idle state that is to be entered. The platform extension plug-in (PEP) previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor-idle-state indexes range from 0 to N-1.

`PlatformState`

[in] The index of the platform idle state that the hardware platform will enter when the processor enters the processor idle state specified by **ProcessorState**. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified M platform idle states, valid platform-idle-state indexes range from 0 to M-1. If no change in platform idle state will occur, this member will contain the value **PEP_PLATFORM_IDLE_STATE_NONE** (0xffffffff).

`VetoReason`

[out] The nonzero index value that identifies reason the PEP needs to veto this idle state transition, or **PEP_IDLE_VETO_NONE** (0) if the PEP is prepared to immediately make this transition. If the PEP supports N veto reasons, valid veto reason indexes range from 1 to N. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification.

Remarks

This structure is used by the PEP_NOTIFY_PPM_TEST_IDLE_STATE notification. The **ProcessorState** and **PlatformState** members contain input values that are supplied by the Windows power management framework (PoFx). The PEP writes an output value to the **VetoReason** member of this structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

PEP_NOTIFY_PPM_TEST_IDLE_STATE

PEP_PREPARE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PREPARE_DEVICE** structure identifies a device that must be started up in preparation for its use by the operating system.

Syntax

```
typedef struct _PEP_PREPARE_DEVICE {
  PCUNICODE_STRING DeviceId;
  BOOLEAN          DeviceAccepted;
} PEP_PREPARE_DEVICE, *PPEP_PREPARE_DEVICE;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`DeviceAccepted`

[out] Whether the PEP claims ownership of the device. The PEP sets this member to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device. The PEP that claims ownership is responsible for handling device power management (DPM) notifications for the device.

Remarks

This structure is used by the PEP_DPM_PREPARE_DEVICE notification. The **DeviceId** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx). The **DeviceAccepted** member contains an output value that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_PREPARE_DEVICE

UNICODE_STRING

PEP_PROCESSOR_FEEDBACK_COUNTER structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_FEEDBACK_COUNTER** structure describes a feedback counter to the operating system.

Syntax

```
typedef struct _PEP_PROCESSOR_FEEDBACK_COUNTER {
  struct {
    ULONG Affinitized : 1;
    ULONG Type : 2;
    ULONG Counter : 4;
    ULONG DiscountIdle : 1;
    ULONG Reserved : 24;
  };
  ULONG  NominalRate;
} PEP_PROCESSOR_FEEDBACK_COUNTER, *PPEP_PROCESSOR_FEEDBACK_COUNTER;
```

Members

`Affinitized`

Identifies the counter process affinity. If set to 1, the counter must be read while executing on the target processor, otherwise, it will be set to 0.

`Type`

Specifies the data type of the counter.

The processor feedback counter data types are:

VALUE	MEANING
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_T** **YPE** **_IN** **STA** **NT** **ANE** **OU** **S** 0x0 0	The feedback counter returns the instantaneous value of the property being counted.

PR **OCE** **SSO** **R_F** **EED** **BAC** **K_T** **YPE** **_RE** **LAT** **IVE** 0x0 1	The feedback counter returns two incrementing values: the nominal count, and the actual count. Nominal count increments at a fixed nominal rate. Actual count increments at a variable rate relative to the property being counted. When the property is incrementing at its nominal rate, the two values should increment at the same rate. To compute an average rate over a time period, the OS reads the counter once and the beginning of the period and once at the end and computes:

`Counter`

Specifies the data the counter is providing.

The processor feedback counter types are:

VALUE	MEANING
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_C** **OU** **NTE** **R_F** **REQ** **UEN** **CY** 0x0 0	The feedback counter returns the clock speed of the processor. The nominal rate is the nominal clock speed, in MHz.
PR **OCE** **SSO** **R_F** **EED** **BAC** **K_C** **OU** **NTE** **R_P** **ERF** **OR** **MA** **NCE** 0x0 1	The feedback counter returns the current processor performance. The nominal rate is equivalent to the processor's **NominalPerformance** (see PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification).

`DiscountIdle`

`Reserved`

This member is reserved and should be set to zero.

`NominalRate`

Specifies the nominal rate of the counter.

Remarks

This structure

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP structures

PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification

PEP_PROCESSOR_IDLE_DEPENDENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_DEPENDENCY** structure specifies the dependencies of a platform idle state on the specified processor.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_DEPENDENCY {
  POHANDLE TargetProcessor;
  UCHAR    ExpectedState;
  BOOLEAN  AllowDeeperStates;
  BOOLEAN  LooseDependency;
} PEP_PROCESSOR_IDLE_DEPENDENCY, *PPEP_PROCESSOR_IDLE_DEPENDENCY;
```

Members

`TargetProcessor`

A **POHANDLE** value that identifies the processor. The platform extension plug-in (PEP) received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`ExpectedState`

The index of the processor idle state that this processor must enter to initiate the platform's transition to this platform idle state. If the **IdleStates** array in the PEP_PPM_QUERY_IDLE_STATES_V2 structure contains N elements, the idle states are numbered 0 to N-1 in the order in which they appear in the array.

`AllowDeeperStates`

Whether a secondary (subordinate) processor can be in a deeper processor idle state than **ExpectedState** and still satisfy the constraints of the transition to the platform idle state. Set this member to **TRUE** if a secondary processor in a deeper idle state can still satisfy the constraints, and to **FALSE** if it cannot. A primary processor (identified by the **TargetProcessor** member) can enter the selected idle state only after its secondary processors have entered their corresponding idle states.

`LooseDependency`

Whether the platform idle state has a loose dependency on the idle state of this processor. Set this member to **TRUE** to indicate a loose dependency in which a best effort to synchronize the idle state transitions of the platform and processor is sufficient. Set to **FALSE** to indicate a dependency in which the operating system is required to strictly synchronize these transitions. If **LooseDependency** is **FALSE**, the **WakesSpuriously** bit in the PEP_PROCESSOR_IDLE_STATE_V2 structure for this processor must be **FALSE**.

Remarks

The **DependencyArray** member of the PEP_PPM_IDLE_SELECT structure is a pointer to an array of **PEP_PROCESSOR_IDLE_DEPENDENCY** structures. The **DependencyArray** member of the PEP_PLATFORM_IDLE_STATE structure is the first element in an array of **PEP_PROCESSOR_IDLE_DEPENDENCY** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_PLATFORM_IDLE_STATE

PEP_PPM_IDLE_SELECT

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_IDLE_STATE_V2

PEP_PROCESSOR_IDLE_STATE_UPDATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure contains the updated properties of a processor idle state.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_STATE_UPDATE {
  ULONG Version;
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PROCESSOR_IDLE_STATE_UPDATE, *PPEP_PROCESSOR_IDLE_STATE_UPDATE;
```

Members

`Version`

The version number of this structure. Set this member to PEP_PROCESSOR_IDLE_STATE_UPDATE_VERSION.

`Latency`

The worst-case latency, in 100-nanosecond units, that the platform requires to wake from this platform idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the platform must spend in this idle state to make a transition to this state worthwhile. PoFx uses this member value as a hint to avoid switching the platform to an idle state unless the platform is likely to remain in this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

The *Update* parameter to the UpdateProcessorIdleState routine is a pointer to a **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure. This routine updates the properties of the specified idle state for a processor. The PEP must not call this routine until it has responded to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification for this processor.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

UpdateProcessorIdleState

PEP_PROCESSOR_IDLE_STATE_V2 structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_IDLE_STATE_V2** structure describes a processor idle state that the platform extension plug-in (PEP) supports.

Syntax

```
typedef struct _PEP_PROCESSOR_IDLE_STATE_V2 {
  union {
    ULONG Ulong;
    struct {
      ULONG Interruptible : 1;
      ULONG CacheCoherent : 1;
      ULONG ThreadContextRetained : 1;
      ULONG CStateType : 4;
      ULONG WakesSpuriously : 1;
      ULONG PlatformOnly : 1;
      ULONG Autonomous : 1;
      ULONG Reserved : 22;
    };
  };
  ULONG Latency;
  ULONG BreakEvenDuration;
} PEP_PROCESSOR_IDLE_STATE_V2, *PPEP_PROCESSOR_IDLE_STATE_V2;
```

Members

`Ulong`

`Interruptible`

`CacheCoherent`

`ThreadContextRetained`

`CStateType`

`WakesSpuriously`

`PlatformOnly`

`Autonomous`

`Reserved`

`Latency`

The worst-case latency, in 100-nanosecond units, that the processor requires to wake from this idle state in response to a wake event.

`BreakEvenDuration`

The minimum amount of time, specified in 100-nanosecond units, that the processor must spend in this idle state to make a transition to this state worthwhile. The Windows power management framework (PoFx) uses this member value as a hint to avoid switching a processor to an idle state unless the processor is likely to remain in

this state for at least the amount of time specified by **BreakEvenDuration**.

Remarks

This structure is used in conjunction with the PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. The **IdleStates** member of the PEP_PPM_QUERY_IDLE_STATES_V2 structure is the first element in an array of **PEP_PROCESSOR_IDLE_STATE_V2** structures.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

PEP_PPM_QUERY_IDLE_STATES_V2

PEP_PROCESSOR_PARK_PREFERENCE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_PREFERENCE** structure indicates the preferences of the operating system and platform extension plug-in (PEP) regarding whether the specified processor should be parked to reduce power consumption.

Syntax

```
typedef struct _PEP_PROCESSOR_PARK_PREFERENCE {
  PEPHANDLE Processor;
  UCHAR     PoPreference;
  UCHAR     PepPreference;
} PEP_PROCESSOR_PARK_PREFERENCE, *PPEP_PROCESSOR_PARK_PREFERENCE;
```

Members

`Processor`

A **PEPHANDLE** value that identifies the processor. The PEP supplied this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`PoPreference`

The operating system's parking preference for this processor. For more information, see Remarks.

`PepPreference`

The PEP's parking preference for this processor. For more information, see Remarks.

Remarks

The **Processors** member of the PEP_PPM_PARK_SELECTION structure is a pointer to an array of **PEP_PROCESSOR_PARK_PREFERENCE** structures.

The **Processor** and **PoPreference** members of the **PEP_PROCESSOR_PARK_PREFERENCE** structure contain input values that are supplied by the Windows power management framework when a PEP_NOTIFY_PPM_PARK_SELECTION notification is sent to the PEP. The **PepPreference** member contains an output value that the PEP writes to the structure in response to this notification.

The following table lists the three values that are defined for the **PoPreference** and **PepPreference** members.

PARKING PREFERENCE	VALUE	MEANING
PROCESSOR_PARK_PREFERENCE_NONE	0x0	No preference. Either parked or unparked is okay.
PROCESSOR_PARK_PREFERENCE_PARKED	0x1	The operating system or PEP specifies that the processor should be parked.

PROCESSOR_PARK_PREFERENCE_UNPARKED	0x2	The operating system or PEP specifies that the processor should not be parked.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_NOTIFY_PPM_PARK_SELECTION

PEP_PPM_PARK_SELECTION

PEP_PROCESSOR_PARK_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_PROCESSOR_PARK_STATE** structure describes the parking state for a single processor.

Syntax

```
typedef struct _PEP_PROCESSOR_PARK_STATE {
  PEPHANDLE Processor;
  BOOLEAN   Parked;
  UCHAR     Reserved[3];
} PEP_PROCESSOR_PARK_STATE, *PPEP_PROCESSOR_PARK_STATE;
```

Members

`Processor`

Specifies the **PEPHANDLE** associated with this processor.

`Parked`

Specifies whether or not this processor is parked.

`Reserved`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_PPM_PARK_MASK notification

PEP_PPM_PARK_MASK

PEP_PROCESSOR_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

Use in the **PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES** notification. This structure describes the properties of a single performance state.

Syntax

```
typedef struct _PEP_PROCESSOR_PERF_STATE {
  ULONG Performance;
  ULONG Frequency;
  ULONG Reserved[4];
} PEP_PROCESSOR_PERF_STATE, *PPEP_PROCESSOR_PERF_STATE;
```

Members

`Performance`

On output, the performance level (on the scale returned by the PEP_QUERY_COMPONENT_PERF_CAPABILITIES notification) of this performance state.

`Frequency`

On output, the frequency (in MHz) of this performance state.

`Reserved`

Reserved.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	pepfx.h (include Pep_x.h)

See Also

PEP_PPM_QUERY_DISCRETE_PERF_STATES

Processor power management (PPM) notifications

PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_CAPABILITIES** structure specifies the number of performance state (P-state) sets that are defined for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_CAPABILITIES {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     SetCount;
} PEP_QUERY_COMPONENT_PERF_CAPABILITIES, *PPEP_QUERY_COMPONENT_PERF_CAPABILITIES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`SetCount`

[out] The number of P-state sets that are defined for this component.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification. The **DeviceHandle** and **Component** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent. The **SetCount** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_SET** structure contains query information about a set of performance state values (P-state set) for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_SET {
  PEPHANDLE           DeviceHandle;
  ULONG               Component;
  ULONG               Set;
  ULONGLONG           Flags;
  PEP_PERF_STATE_UNIT Unit;
  PEP_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG Count;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PEP_QUERY_COMPONENT_PERF_SET, *PPEP_QUERY_COMPONENT_PERF_SET;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If this component has M P-state sets, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`Flags`

[in] A set of input flags. No flag bits are currently defined for this member, which is always zero.

`Unit`

[out] A PEP_PERF_STATE_UNIT enumeration value that indicates whether the values for this P-state set are expressed in hertz (frequency units) or bits per second (bandwidth units).

`Type`

[out] A PEP_PERF_STATE_TYPE enumeration value that indicates whether the performance values for this P-state set are expressed as a list of discrete values or as a continuous range of values.

`Discrete`

`Discrete.Count`

`Range`

`Range.Minimum`

`Range.Maximum`

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_SET notification. The **DeviceHandle**, **Component**, **Set**, and **Flags** members contain input values that are supplied by the Windows power management framework (PoFx) when this notification is sent. The **Discrete** member or **Range** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_SET

PEP_DPM_REGISTER_DEVICE

PEP_PERF_STATE_TYPE

PEP_PERF_STATE_UNIT

PEP_QUERY_COMPONENT_PERF_SET_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_SET_NAME** structure contains query information about a set of performance state values (P-state set) for a component.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_SET_NAME {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     Set;
  USHORT    NameSize;
  PWCHAR    Name;
} PEP_QUERY_COMPONENT_PERF_SET_NAME, *PPEP_QUERY_COMPONENT_PERF_SET_NAME;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If this component has M P-state sets, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`NameSize`

[in, out] On input, the size, in bytes, of the buffer pointed to by the **Name** member. If **Name** is NULL, the PEP overwrites the input value of **NameSize** with the buffer size required for the name string.

`Name`

[in] A pointer to an output buffer. The PEP writes the name of the P-state to this buffer. The name is stored as a wide-character, null-terminated string. The **Name** member is NULL if the Windows power management framework (PoFx) needs to determine how large a buffer to allocate for the name string. If **Name** is non-NULL, the buffer must be large enough to contain the entire string, including the terminating null character.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME notification. The

DeviceHandle, **Component**, and **Set** members of the structure contain input values supplied by PoFx when this notification is sent. If the **Name** member is non-NULL, the PEP writes a string to the buffer pointed to by **Name**. If **Name** is NULL, PEP writes the required buffer size to the **NameSize** member.

The string that the PEP writes to the output buffer should contain a descriptive name for the P-state set. This name is intended to make log entries and diagnostic messages easier to understand.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES

PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_COMPONENT_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_COMPONENT_PERF_STATES** structure contains a list of discrete performance state (P-state) values for the specified P-state set.

Syntax

```
typedef struct _PEP_QUERY_COMPONENT_PERF_STATES {
  PEPHANDLE       DeviceHandle;
  ULONG           Component;
  ULONG           Set;
  PPEP_PERF_STATE States;
} PEP_QUERY_COMPONENT_PERF_STATES, *PPEP_QUERY_COMPONENT_PERF_STATES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If M is the number of P-state sets for this component, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`States`

[in] A pointer to an output buffer. The PEP writes an array of PEP_PERF_STATE structures to this buffer. Each array element describes one P-state in the P-state set specified by the **Set** member. The Windows power management framework (PoFx) allocated this buffer, which is guaranteed to be large enough to contain an array of the length that the PEP previously wrote to the **Discrete.Count** member of the PEP_QUERY_COMPONENT_PERF_SET structure in response to a PEP_DPM_QUERY_COMPONENT_PERF_SET notification.

Remarks

This structure is used by the PEP_DPM_QUERY_COMPONENT_PERF_STATES notification. All four members of this structure contain input values that are supplied by PoFx when this notification is sent.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_COMPONENT_PERF_STATES

PEP_DPM_REGISTER_DEVICE

PEP_PERF_STATE

PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_CURRENT_COMPONENT_PERF_STATE** structure contains information about the current P-state in the specified P-state set.

Syntax

```
typedef struct _PEP_QUERY_CURRENT_COMPONENT_PERF_STATE {
  PEPHANDLE DeviceHandle;
  ULONG     Component;
  ULONG     Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PEP_QUERY_CURRENT_COMPONENT_PERF_STATE, *PPEP_QUERY_CURRENT_COMPONENT_PERF_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Set`

[in] The index that identifies this P-state set. If M is the number of P-state sets for this component, P-state set indexes range from 0 to M–1. The PEP previously specified the number of P-state sets in response to a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification.

`StateIndex`

`StateValue`

Remarks

This structure is used by the PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE notification. The **DeviceHandle**, **Component**, and **Set** members contain input values supplied by the Windows power management framework (PoFx) when this notification is sent. The **StateIndex** member or **StateValue** member contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE

PEP_DPM_REGISTER_DEVICE

PEP_QUERY_SOC_SUBSYSTEM structure

1/30/2019 • 3 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM** structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM notification to gather basic information about a particular system on a chip (SoC) subsystem.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM {
  ULONG          PlatformIdleStateIndex;
  ULONG          SubsystemIndex;
  PVOID          SubsystemHandle;
  UNICODE_STRING ParentName;
  UNICODE_STRING SubsystemName;
  ULONG          MetadataCount;
  ULONG          Flags;
} PEP_QUERY_SOC_SUBSYSTEM, *PPEP_QUERY_SOC_SUBSYSTEM;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index that is being queried by the kernel.

`SubsystemIndex`

[in] The subsystem index, which is a unique index assigned by the OS to each subsystem within the context of a given **PlatformIdleStateIndex**.

The kernel initializes this value to zero and increments it for each subsequent subsystem static info notification. A value of 0 means it's the first such notification for the given **PlatformIdleStateIndex**. This field is intended to give the PEP context between instances of this notification. It is the only field whose input value varies across instances of this notification for a given **PlatformIdleStateIndex** and a PEP might use this value to look up a subsystem.

The PEP can ignore this value.

`SubsystemHandle`

[out] A context pointer that the PEP can optionally assign a value to. It will receive the pointer on subsequent notifications for this particular subsystem. In subsequent notifications, a PEP uses **PlatformIdleStateIndex** along with **SubsystemName** and/or **SubsystemHandle** to look up a particular subsystem's accounting data.

By default the value if this member is zero. This value is ignored by the OS and used only in notifications.

The PEP is not required to fill this member.

`ParentName`

[in/out] A buffer for holding the parent name of the corresponding subsystem. Each subsystem has a parent and subsystems with the same value for **ParentName** are considered siblings. Subsystem sibling hierarchies are optional. In the absence of any hierarchy, all subsystems are top level subsystems and specify a common value for **ParentName**. To indicate a hierarchy, top level subsystems specify a common **ParentName** while each non-top-level subsystem specifies its parent subsystem's **SubsystemName** for **ParentName**.

Prior to entry, the kernel will:

- Pre-allocate 64 **WCHARs** for **ParentName.Buffer[]**
- Zero the buffer
- Set **ParentName.MaximumLength** to the size of the buffer in bytes, and
- Set **ParentName.Length** = **0**

Prior to exit, the PEP must copy a null-terminated string into **ParentName.Buffer** and update **ParentName.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **ParentName.Buffer** to provide the parent name. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the parent name, if necessary, so that it does not exceed the length specified in **ParentName.MaximumLength** (including the terminating **UNICODE_NULL** character).

`SubsystemName`

[in/out] A buffer for holding this subsystem's name. Every subsystem has a subsystem name and **SubsystemName** must be unique among all subsystems within the context of a given platform idle state. A subsystem's **SubsystemName** cannot be the same as **ParentName**.

Prior to entry, the kernel will:

- Pre-allocate 64 **WCHARs** for **SubsystemName.Buffer[]**
- Zero the buffer
- Set **SubsystemName.MaximumLength** to the size of the buffer in bytes, and
- Set **SubsystemName.Length** = **0**

Prior to exit, the PEP must copy a null-terminated string into **SubsystemName.Buffer** and update **SubsystemName.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **SubsystemName.Buffer** to provide the subsystem name. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the subsystem name, if necessary, so that it does not exceed the length specified in **SubsystemName.MaximumLength** (including the terminating **UNICODE_NULL** character).

`MetadataCount`

[out] The number of metadata key/value string pairs tallied by this SoC subsystem.

Metadata is optional. The PEP returns zero if it does not report any metadata for this SoC subsystem.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM notification

PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME** structure is used by the PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification to collect details about the blocking duration for a particular system on a chip (SoC) subsystem.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME {
  ULONG            PlatformIdleStateIndex;
  PVOID            SubsystemHandle;
  PCUNICODE_STRING SubsystemName;
  ULONG64          BlockingTime;
  ULONG            Flags;
} PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME, *PPEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for the SoC subsystem that the OS is querying.

`SubsystemHandle`

[in] A context pointer that the PEP previously provided on subsystem initialization. The context pointer is optional, so if none was provided then the value will be zero. The PEP is free to ignore this field.

`SubsystemName`

[in] The name of the subsystem whose blocking time is being queried. This name was provided by the PEP on subsystem initialization.

`BlockingTime`

[out] The blocking time tallied for this subsystem in 100 nanosecond units since the previous PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification

PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification

PEP_QUERY_SOC_SUBSYSTEM_COUNT structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_COUNT** structure is used to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_COUNT {
  ULONG PlatformIdleStateIndex;
  ULONG SubsystemCount;
  ULONG Flags;
} PEP_QUERY_SOC_SUBSYSTEM_COUNT, *PPEP_QUERY_SOC_SUBSYSTEM_COUNT;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for which the kernel is querying about.

`SubsystemCount`

[out] The number of SoC subsystems tallied by the PEP for the specified platform idle state. The PEP cannot return 0 in this parameter; instead the PEP should return **FALSE** to the PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification that provided this structure.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification

PEP_QUERY_SOC_SUBSYSTEM_METADATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_QUERY_SOC_SUBSYSTEM_METADATA** structure is used with the **PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA** notification to collect optional metadata about the system on a chip (SoC) subsystem whose blocking time has just been queried.

Syntax

```
typedef struct _PEP_QUERY_SOC_SUBSYSTEM_METADATA {
  ULONG                       PlatformIdleStateIndex;
  PVOID                       SubsystemHandle;
  PCUNICODE_STRING            SubsystemName;
  ULONG                       Flags;
  ULONG                       MetadataCount;
  PPEP_SOC_SUBSYSTEM_METADATA Metadata[ANYSIZE_ARRAY];
} PEP_QUERY_SOC_SUBSYSTEM_METADATA, *PPEP_QUERY_SOC_SUBSYSTEM_METADATA;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for for the SoC subsystem that the OS is querying.

`SubsystemHandle`

[in] A context pointer that the PEP previously provided on subsystem initialization. The context pointer is optional, so if none was provided then the value will be zero. The PEP is free to ignore this field.

`SubsystemName`

[in] The name of the subsystem whose metadata is being queried.

`Flags`

This member is reserved and should be set to zero.

`MetadataCount`

[in] The number of entries in the **Metadata** array. The PEP previously provided this value as **PEP_QUERY_SOC_SUBSYSTEM.MetadataCount**.

`Metadata`

[in/out] An array of pointers to PEP_SOC_SUBSYSTEM_METADATA structures. Each entry holds one key/value metadata string-pair.

Requirements

Minimum supported client	Supported starting with Windows 10.

Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA

PEP_SOC_SUBSYSTEM_METADATA

PEP_REGISTER_COMPONENT_PERF_STATES structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_COMPONENT_PERF_STATES** structure describes the performance states (P-states) of the specified component.

Syntax

```
typedef struct _PEP_REGISTER_COMPONENT_PERF_STATES {
  PEPHANDLE                DeviceHandle;
  ULONG                    Component;
  ULONGLONG                Flags;
  PPEP_COMPONENT_PERF_INFO PerfStateInfo;
} PEP_REGISTER_COMPONENT_PERF_STATES, *PPEP_REGISTER_COMPONENT_PERF_STATES;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The platform extension plug-in (PEP) supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`Flags`

[in] A set of input flags. No flag bits are currently defined for this member, which is always zero.

`PerfStateInfo`

[in] A pointer to a PEP_COMPONENT_PERF_INFO structure that contains the performance information for this component's P-states. If this member is non-NULL, it points to P-state information that was supplied by the device driver.

Remarks

This structure is used by the PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification. The values of all members of the **PEP_REGISTER_COMPONENT_PERF_STATES** structure are supplied by the Windows power management framework (PoFx) when the notification is sent. The PEP must not write to this structure.

If the **PerfStateInfo** member is not NULL, the device driver has provided P-state information and requires the PEP to use it.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_COMPONENT_PERF_INFO

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_COMPONENT_PERF_STATES

PEP_DPM_REGISTER_DEVICE

PEP_REGISTER_CRASHDUMP_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_CRASHDUMP_DEVICE** structure provides a callback routine to turn on a crash-dump device.

Syntax

```
typedef struct _PEP_REGISTER_CRASHDUMP_DEVICE {
  PPEPCALLBACKPOWERONCRASHDUMPDEVICE PowerOnDumpDeviceCallback;
  PEPHANDLE                          DeviceHandle;
} PEP_REGISTER_CRASHDUMP_DEVICE, *PPEP_REGISTER_CRASHDUMP_DEVICE;
```

Members

`PowerOnDumpDeviceCallback`

[out] A pointer to a PowerOnDumpDeviceCallback callback routine that is implemented by the platform extension plug-in (PEP). This routine handles requests from the Windows kernel to turn on the crash-dump device so that a crash dump can be saved. For more information, see Kernel-Mode Dump Files.

`DeviceHandle`

[in] A PEPHANDLE value that identifies the crash-dump device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_REGISTER_CRASHDUMP_DEVICE notification. The **DeviceHandle** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx) when this notification is sent to the PEP. The **PowerOnDumpDeviceCallbackmember** contains an output value that the PEP writes to the structure in response to the notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_CRASHDUMP_DEVICE

PEP_DPM_REGISTER_DEVICE

PowerOnDumpDeviceCallback

PEP_REGISTER_DEBUGGER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_DEBUGGER** structure identifies a registered device that is a core system resource that provides debugger transport.

Syntax

```
typedef struct _PEP_REGISTER_DEBUGGER {
  PEPHANDLE DeviceHandle;
} PEP_REGISTER_DEBUGGER, *PPEP_REGISTER_DEBUGGER;
```

Members

`DeviceHandle`

[in] A **PEPHANDLE** value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_REGISTER_DEBUGGER notification. The **DeviceHandle** member contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEBUGGER

PEP_DPM_REGISTER_DEVICE

PEP_REGISTER_DEVICE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REGISTER_DEVICE_V2** structure describes a device whose driver stack has just registered with the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_REGISTER_DEVICE_V2 {
  PCUNICODE_STRING           DeviceId;
  POHANDLE                   KernelHandle;
  PPEP_DEVICE_REGISTER_V2    Register;
  PEPHANDLE                  DeviceHandle;
  PEP_DEVICE_ACCEPTANCE_TYPE DeviceAccepted;
} PEP_REGISTER_DEVICE_V2, *PPEP_REGISTER_DEVICE_V2;
```

Members

`DeviceId`

[in] A string that uniquely identifies the device. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`KernelHandle`

[in] A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Register`

[in] A pointer to a PEP_DEVICE_REGISTER_V2 structure that describes the power management attributes of all the components in the device. For more information, see Remarks.

`DeviceHandle`

[out] A PEPHANDLE value that the PEP creates to identify this device. PoFx will use this handle to identify the device in future device power management (DPM) notifications.

`DeviceAccepted`

[out] A PEP_DEVICE_ACCEPTANCE_TYPE enumeration value that indicates whether the PEP claims ownership of the device. The PEP that claims ownership is responsible for handling DPM notifications for the device.

Remarks

This structure is used by the PEP_DPM_REGISTER_DEVICE notification. The first three members of this structure contain input values supplied by PoFx. The last two members contain output values that the PEP writes to the structure in response to this notification.

The **Register** member contains a pointer to an input buffer allocated by PoFx. PoFx writes the PEP_DEVICE_REGISTER_V2 structure and associated data to this structure before sending the PEP_DPM_REGISTER_DEVICE notification to the PEP. The contents of this buffer remain valid only until the PEP

finishes handling the notification and returns from the AcceptDeviceNotification callback.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

AcceptDeviceNotification

PEP_DEVICE_ACCEPTANCE_TYPE

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

UNICODE_STRING

PEP_REQUEST_COMPONENT_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_REQUEST_COMPONENT_PERF_STATE** structure contains a list of performance state (P-state) changes requested by the Windows power management framework (PoFx), plus status information about the handling of these requests by the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_REQUEST_COMPONENT_PERF_STATE {
  PEPHANDLE                          DeviceHandle;
  ULONG                              Component;
  BOOLEAN                            Completed;
  BOOLEAN                            Succeeded;
  ULONG                              PerfRequestsCount;
  PPEP_COMPONENT_PERF_STATE_REQUEST PerfRequests;
} PEP_REQUEST_COMPONENT_PERF_STATE, *PPEP_REQUEST_COMPONENT_PERF_STATE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Completed`

[out] Whether the PEP has completed the requested P-state changes. Set to TRUE to indicate that the P-state changes completed synchronously. Set to FALSE to indicate that the requested P-state changes are pending and will be completed asynchronously by a PEP worker thread. For more information, see Remarks.

`Succeeded`

[out] Whether the requested P-state changes successfully completed. Set to TRUE if the PEP successfully completed all P-state changes requested in the **PerfRequests** array. Set to FALSE if the PEP was unable to perform all the requested P-state changes, in which case the P-states in the hardware were left unchanged. The **Succeeded** member is used only for *synchronous* P-state changes. If the PEP sets **Completed** to FALSE, PoFx ignores the value in the **Succeeded** member.

`PerfRequestsCount`

[in] The number of P-state changes requested in the array pointed to by the **PerfRequests** member.

`PerfRequests`

[in] A pointer to an array of PEP_COMPONENT_PERF_STATE_REQUEST structures. Each element in the array specifies a new performance level to assign to a P-state set. The number of array elements is specified by the

PerfRequestsCount member.

Remarks

This structure is used by the PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification. The **DeviceHandle**, **Component**, **PerfRequestsCount**, and **PerfRequests** members of the structure contain input values that are supplied by PoFx when this notification is sent. The **Completed** and **Succeeded** members contain output values that the PEP writes to the structure in response to the notification.

The PEP can complete the requested P-state changes either synchronously or asynchronously. The PEP can set the **Completed** member to TRUE to indicate that all the requests in the **PEP_DPM_REQUEST_COMPONENT_PERF_STATE** notification have been completed synchronously—that is, before the return from the PEP's AcceptDeviceNotification callback routine. Or, the PEP can set **Completed** to FALSE to indicate that the requests will be completed asynchronously. To complete the requests asynchronously, the PEP should call the RequestWorker routine to inform PoFx that the PEP has a work request to submit, and PoFx will respond by sending a PEP_DPM_WORK notification to the PEP.

PoFx writes the requested P-state changes to the **PerfRequests** array before the **PEP_DPM_REQUEST_COMPONENT_PERF_STATE** notification is sent. If the PEP handles the requested P-state changes *synchronously*, the contents of the **PerfRequests** array remain valid only until the return from the *AcceptDeviceNotification* callback routine. If the requests are completed *asynchronously*, the array contents remain valid until the PEP calls the CompleteWork routine to notify PoFx that the PEP has completed the pending requests.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

AcceptDeviceNotification

PEP_COMPONENT_PERF_STATE_REQUEST

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_DPM_WORK

RequestWorker

PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING** structure is provided to the platform extension plug-in (PEP) as part of a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification.

Syntax

```
typedef struct _PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING {
  ULONG PlatformIdleStateIndex;
  ULONG Flags;
} PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING, *PPEP_RESET_SOC_SUBSYSTEM_ACCOUNTING;
```

Members

`PlatformIdleStateIndex`

[in] The platform idle state index for the subsystem accounting to be reset.

`Flags`

This member is reserved and should be set to zero.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification

PEP_SOC_SUBSYSTEM_METADATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_SOC_SUBSYSTEM_METADATA** structure contains key-value pairs that contain metadata for a system on a chip (SoC) subsystem. It is used in the context of a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification sent to a platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_SOC_SUBSYSTEM_METADATA {
  UNICODE_STRING Key;
  UNICODE_STRING Value;
} PEP_SOC_SUBSYSTEM_METADATA, *PPEP_SOC_SUBSYSTEM_METADATA;
```

Members

`Key`

[in/out] A buffer for the PEP to write the key portion of the metadata string-pair. **Key** must be unique among all **Key** values reported by this subsystem.

Prior to entry into the notification callback routine, the kernel will:

- Pre-allocate 64 **WCHARs** for **Key.Buffer[]**
- Zero the buffer
- Set **Key.MaximumLength** to the size of the buffer in bytes, and
- Set **Key.Length** = **0**

Prior to exit from the callback routine, the PEP must copy a null-terminated string into **Key.Buffer** and update **Key.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **Key.Buffer** to provide the key. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the key string, if necessary, so that it does not exceed the length specified in **Key.MaximumLength** (including the terminating **UNICODE_NULL** character).

`Value`

[in/out] A buffer for the PEP to write the value portion of the metadata string-pair.

Prior to entry into the notification callback routine, the kernel will:

- Pre-allocate 64 **WCHARs** for **Value.Buffer[]**
- Zero the buffer
- Set **Value.MaximumLength** to the size of the buffer in bytes, and
- Set **Value.Length** = **0**

Prior to exit from the callback routine, the PEP must copy a null-terminated string into **Value.Buffer** and update **Value.Length** with a byte count that is **sizeof(WCHAR)** times the number of UNICODE characters copied, not including the terminating **UNICODE_NULL**.

The PEP must use the allocated memory that is pointed to by the address in **Value.Buffer** to provide the metadata value. Since this memory is pre-allocated, its size cannot be changed. The PEP is responsible for truncating the value string, if necessary, so that it does not exceed the length specified in **Value.MaximumLength** (including the terminating **UNICODE_NULL** character).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA

PEP_QUERY_SOC_SUBSYSTEM_METADATA

PEP_SYSTEM_LATENCY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_SYSTEM_LATENCY** structure specifies the new value for the system latency tolerance.

Syntax

```
typedef struct _PEP_SYSTEM_LATENCY {
  ULONGLONG Latency;
} PEP_SYSTEM_LATENCY, *PPEP_SYSTEM_LATENCY;
```

Members

`Latency`

[in] The overall system latency tolerance, in 100-nanosecond units. This member specifies the maximum latency that the operating system can tolerate in the time required to move a component from a low-power F*x* state to F0.

Remarks

This structure is used by the PEP_DPM_SYSTEM_LATENCY_UPDATE notification. The **Latency** member of the structure contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_SYSTEM_LATENCY_UPDATE

PEP_UNMASKED_INTERRUPT_FLAGS union

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_UNMASKED_INTERRUPT_FLAGS** union indicates whether an unmasked interrupt source is a primary interrupt or a secondary interrupt.

Syntax

```
typedef union _PEP_UNMASKED_INTERRUPT_FLAGS {
  struct {
    USHORT SecondaryInterrupt : 1;
    USHORT Reserved : 15;
  };
  USHORT AsUSHORT;
} PEP_UNMASKED_INTERRUPT_FLAGS, *PPEP_UNMASKED_INTERRUPT_FLAGS;
```

Members

`SecondaryInterrupt`

`Reserved`

`AsUSHORT`

A USHORT value that contains all of the unmasked interrupt flags.

Remarks

The **Flags** member of the PEP_UNMASKED_INTERRUPT_INFORMATION structure is a **PEP_UNMASKED_INTERRUPT_FLAGS** union.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_UNMASKED_INTERRUPT_INFORMATION

PEP_UNMASKED_INTERRUPT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_UNMASKED_INTERRUPT_INFORMATION** structure contains information about an interrupt source.

Syntax

```
typedef struct _PEP_UNMASKED_INTERRUPT_INFORMATION {
  USHORT                       Version;
  USHORT                       Size;
  PEP_UNMASKED_INTERRUPT_FLAGS Flags;
  KINTERRUPT_MODE              Mode;
  KINTERRUPT_POLARITY          Polarity;
  ULONG                        Gsiv;
  USHORT                       PinNumber;
  PEPHANDLE                    DeviceHandle;
} PEP_UNMASKED_INTERRUPT_INFORMATION, *PPEP_UNMASKED_INTERRUPT_INFORMATION;
```

Members

`Version`

The version of this structure.

`Size`

The size, in bytes, of this structure.

`Flags`

A PEP_UNMASKED_INTERRUPT_FLAGS union that indicates whether the interrupt is a primary or secondary interrupt. For more information, see Primary and Secondary Interrupts.

`Mode`

A KINTERRUPT_MODE enumeration value. This member indicates whether the interrupt is edge-triggered or level-triggered.

`Polarity`

A KINTERRUPT_POLARITY enumeration value. This member indicates which edge or level of the interrupt signal triggers the interrupt.

`Gsiv`

The global system interrupt vector (GSIV) number that identifies this interrupt. The ACPI firmware assigns GSIV numbers to all primary interrupt lines. For secondary (GPIO) interrupt lines, the GSIV number is dynamically assigned by the operating system.

`PinNumber`

For secondary interrupt sources, this member identifies the number of the pin on the general-purpose I/O (GPIO)

controller that is connected to the interrupt signal line from the interrupting device. For primary interrupt sources, this member is undefined.

If a GPIO controller has N GPIO pins, the pins are numbered 0 to N−1. One or more of these GPIO pins might be configured as interrupt inputs.

`DeviceHandle`

For secondary interrupt sources, this member contains the PEP device handle for the GPIO controller that is the source for this interrupt. For primary interrupt sources, this field is undefined.

Remarks

This structure is used by the EnumerateInterruptSource callback routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

EnumerateInterruptSource

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_UNMASKED_INTERRUPT_FLAGS

PEP_UNREGISTER_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_UNREGISTER_DEVICE** structure identifies a device whose registration is being removed from the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_UNREGISTER_DEVICE {
  PEPHANDLE DeviceHandle;
} PEP_UNREGISTER_DEVICE, *PPEP_UNREGISTER_DEVICE;
```

Members

`DeviceHandle`

[in] A PEPHANDLE value that identifies the device. The PEP supplied this handle in response to a previous PEP_DPM_REGISTER_DEVICE notification.

Remarks

This structure is used by the PEP_DPM_UNREGISTER_DEVICE notification. The **DeviceHandle** member contains an input value that is supplied by the Windows power management framework (PoFx).

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_REGISTER_DEVICE

PEP_DPM_UNREGISTER_DEVICE

PEP_WORK structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK** structure indicates whether the PEP has a work request to submit to the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_WORK {
  PPEP_WORK_INFORMATION WorkInformation;
  BOOLEAN               NeedWork;
} PEP_WORK, *PPEP_WORK;
```

Members

`WorkInformation`

[out] A pointer to a PEP-allocated PEP_WORK_INFORMATION structure that describes the work that the PEP is requesting. If **NeedWork** is TRUE, **WorkInformation** must point to a valid **PEP_WORK_INFORMATION** structure. If **NeedWork** is FALSE, **WorkInformation** must be NULL.

`NeedWork`

[out] Whether the PEP has a work request to submit. Set this member to TRUE if the PEP has a work request, or to FALSE if the PEP has no work to request.

Remarks

This structure is used by the PEP_DPM_WORK notification. Both members of the structure contain values that the PEP writes to the structure in response to this notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_WORK

PEP_WORK_INFORMATION

PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE** structure contains the results of an ACPI control method that was asynchronously evaluated by the platform extension plug-in (PEP).

Syntax

```
typedef struct _PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE {
  POHANDLE              DeviceHandle;
  ULONG                 CompletionFlags;
  NTSTATUS              MethodStatus;
  PVOID                 CompletionContext;
  SIZE_T                OutputArgumentSize;
  PACPI_METHOD_ARGUMENT OutputArguments;
} PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE, *PPEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE;
```

Members

`DeviceHandle`

A POHANDLE value that represents the registration of the device with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification in which the PEP elected to provide ACPI services for the device.

`CompletionFlags`

A set of completion flags. No flags are currently defined for this member, which is always set to PEP_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE_FLAG_NONE (0x0).

`MethodStatus`

`CompletionContext`

A pointer to a completion context value. This is the same pointer that PoFx passed to the PEP in the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification that initiated the asynchronous evaluation of the ACPI control method.

`OutputArgumentSize`

The size, in bytes, of the buffer pointed to by the **OutputArguments** member.

`OutputArguments`

A pointer to a buffer to which the PEP has written the result of evaluating the ACPI control method. This buffer starts with a ACPI_METHOD_ARGUMENT structure, which might be followed by additional output data. PoFx allocated this buffer and passed the buffer pointer to the PEP in the PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification that initiated the asynchronous evaluation of the ACPI control method.

Remarks

The **ControlMethodComplete** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE** structure.

A **PEP_WORK_INFORMATION** structure is included in the input parameters to the CompleteWork routine. The PEP writes the control method results to the **ControlMethodComplete** member of this structure before calling the routine. Evaluation of the control method was initiated by a previous PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ACPI_METHOD_ARGUMENT

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_ACPI_NOTIFY structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_ACPI_NOTIFY** structure contains the ACPI Notify code for a device that has generated a hardware event.

Syntax

```
typedef struct _PEP_WORK_ACPI_NOTIFY {
  POHANDLE DeviceHandle;
  ULONG    NotifyCode;
} PEP_WORK_ACPI_NOTIFY, *PPEP_WORK_ACPI_NOTIFY;
```

Members

`DeviceHandle`

A POHANDLE value that represents the registration of the device with PoFx. The platform extension plug-in (PEP) previously received this handle from the Windows power management framework (PoFx) during the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification in which the PEP elected to provide ACPI services for the device.

`NotifyCode`

The Notify code. For a list of Notify codes and their meanings, see section 5.6.6, "Device Object Notifications", of the ACPI 5.0 specification.

Remarks

The **AcpiNotify** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_ACPI_NOTIFY** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_NOTIFY_ACPI_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_COMPLETE_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_COMPLETE_IDLE_STATE** structure identifies a component that the platform extension plug-in (PEP) has prepared for a transition to a new F*x* power state.

Syntax

```
typedef struct _PEP_WORK_COMPLETE_IDLE_STATE {
  POHANDLE DeviceHandle;
  ULONG    Component;
} PEP_WORK_COMPLETE_IDLE_STATE, *PPEP_WORK_COMPLETE_IDLE_STATE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Remarks

The **CompleteIdleState** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_COMPLETE_IDLE_STATE** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PEP_WORK_COMPLETE_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_COMPLETE_PERF_STATE** structure describes the completion status of a previously requested update to the performance values assigned to a list of performance state (P-state) sets.

Syntax

```
typedef struct _PEP_WORK_COMPLETE_PERF_STATE {
  POHANDLE DeviceHandle;
  ULONG    Component;
  BOOLEAN  Succeeded;
} PEP_WORK_COMPLETE_PERF_STATE, *PPEP_WORK_COMPLETE_PERF_STATE;
```

Members

`DeviceHandle`

A handle that represents the registration of the device with the Windows power management framework (PoFx). The PEP received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Succeeded`

Whether the requested P-state changes successfully completed. Set to TRUE if the PEP successfully completed all P-state changes requested in a previous PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification. Set to FALSE if the PEP was unable to perform all the requested P-state changes, in which case the P-states in the hardware were left unchanged.

Remarks

The **CompletePerfState** member of the PEP_WORK_INFORMATION structure is a **PEP_WORK_COMPLETE_PERF_STATE** structure. If PoFx sends a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification to request performance level changes to one or more P-state sets, and the PEP chooses to handle this request asynchronously, the PEP uses a **PEP_WORK_COMPLETE_PERF_STATE** structure to describe the completion status of the request.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_WORK_INFORMATION

PEP_WORK_INFORMATION structure

2/5/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_INFORMATION** structure describes a work item that the PEP is submitting to the Windows power management framework (PoFx).

Syntax

```
typedef struct _PEP_WORK_INFORMATION {
  PEP_WORK_TYPE WorkType;
  union {
    PEP_WORK_POWER_CONTROL                          PowerControl;
    PEP_WORK_COMPLETE_IDLE_STATE                    CompleteIdleState;
    PEP_WORK_COMPLETE_PERF_STATE                    CompletePerfState;
    PEP_WORK_ACPI_NOTIFY                            AcpiNotify;
    PEP_WORK_ACPI_EVALUATE_CONTROL_METHOD_COMPLETE ControlMethodComplete;
  };
} PEP_WORK_INFORMATION, *PPEP_WORK_INFORMATION;
```

Members

`WorkType`

A PEP_WORK_TYPE enumeration value. This member indicates the type of work requested by the PEP, which also determines the type of structure that is contained in the unnamed union in the **PEP_WORK_INFORMATION** structure.

`PowerControl`

`CompleteIdleState`

`CompletePerfState`

`AcpiNotify`

`ControlMethodComplete`

Remarks

The **WorkInformation** member of the PEP_WORK structure is a pointer to a **PEP_WORK_INFORMATION** structure.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_WORK

PEP_WORK

PEP_WORK_ACPI_NOTIFY

PEP_WORK_ACTIVE_COMPLETE

PEP_WORK_COMPLETE_IDLE_STATE

PEP_WORK_COMPLETE_PERF_STATE

PEP_WORK_DEVICE_IDLE

PEP_WORK_DEVICE_POWER

PEP_WORK_IDLE_STATE

PEP_WORK_POWER_CONTROL

PEP_WORK_TYPE

PEP_WORK_POWER_CONTROL structure

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_POWER_CONTROL** structure contains the parameters for a power control request that the platform extension plug-in (PEP) sends directly to a processor driver.

Syntax

```
typedef struct _PEP_WORK_POWER_CONTROL {
  POHANDLE DeviceHandle;
  LPCGUID  PowerControlCode;
  PVOID    RequestContext;
  PVOID    InBuffer;
  SIZE_T   InBufferSize;
  PVOID    OutBuffer;
  SIZE_T   OutBufferSize;
} PEP_WORK_POWER_CONTROL, *PPEP_WORK_POWER_CONTROL;
```

Members

`DeviceHandle`

A POHANDLE value that identifies the processor. The PEP received this handle in a previous PEP_DPM_REGISTER_DEVICE notification.

`PowerControlCode`

[in] A pointer to a GUID value that specifies the power control operation to perform.

`RequestContext`

A pointer to a PEP-defined context value. The PEP might use this value to uniquely identify a power control request message (in case the PEP issues multiple requests with the same control code to the same device).

`InBuffer`

[in] A pointer to a driver-allocated input buffer that contains the input parameters for this power control operation.

`InBufferSize`

[in] The size in bytes of the buffer pointed to by **InBuffer**.

`OutBuffer`

[in] A pointer to a driver-allocated output buffer to which the PEP writes the results of this power control operation.

`OutBufferSize`

[in] The size in bytes of the buffer pointed to by **OutBuffer**.

Remarks

The **PowerControl** member of the PEP_WORK_INFORMATION structure is a pointer to a **PEP_WORK_POWER_CONTROL** structure.

The PEP issues a power control request from a worker thread. The Windows power management framework (PoFx) forwards this request by calling the processor driver's PowerControlCallback routine, if the driver implements this routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

GUID

PEP_DPM_REGISTER_DEVICE

PEP_WORK_INFORMATION

PoFxRegisterDevice

PowerControlCallback

PEP_WORK_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PEP_WORK_TYPE** enumeration describes the type of work that the platform extension plug-in (PEP) is requesting.

Syntax

```
typedef enum _PEP_WORK_TYPE {
  PepWorkRequestPowerControl,
  PepWorkCompleteIdleState,
  PepWorkCompletePerfState,
  PepWorkAcpiNotify,
  PepWorkAcpiEvaluateControlMethodComplete,
  PepWorkMax
} PEP_WORK_TYPE, *PPEP_WORK_TYPE;
```

Constants

PepWorkRequestPowerControl	A request for the device driver to perform a custom power-control operation that uses a device-specific context that the PEP provides for the operation. The driver handles this request in its PowerControlCallback routine.
PepWorkCompleteIdleState	A notification to PoFx that the PEP has asynchronously completed the transition of a component to an idle state. PoFx previously initiated this transition by sending a PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE notification to the PEP.
PepWorkCompletePerfState	A notification to PoFx that the PEP has asynchronously completed the transition of a component to a P-state. PoFx previously initiated this transition by sending a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification to the PEP.
PepWorkAcpiNotify	An ACPI Notify code to describe to PoFx a hardware event that the specified device has generated.
PepWorkAcpiEvaluateControlMethodComplete	A notification to PoFx that the PEP has asynchronously finished evaluating an ACPI control method. PoFx previously initiated the evaluation of this method by sending a PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification to the PEP.
PepWorkMax	Reserved for use by operating system.

Remarks

This enumeration is used by the PEP_WORK_INFORMATION structure to describe the type of work item that the PEP is requesting.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

PEP_WORK_INFORMATION

PowerControlCallback

PO_FX_CORE_DEVICE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_CORE_DEVICE** structure contains information about the power-state attributes of the components in a core system resource, and provides a software interface for power-managing these components.

Syntax

```
typedef struct _PO_FX_CORE_DEVICE {
  ULONG                                        Version;
  ULONG                                        ComponentCount;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK    ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK      ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK ComponentCriticalTransitionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK          ComponentIdleStateCallback;
  PVOID                                        DeviceContext;
  PO_FX_COMPONENT                              Components[ANYSIZE_ARRAY];
} PO_FX_CORE_DEVICE, *PPO_FX_CORE_DEVICE;
```

Members

`Version`

The version number of this structure.

`ComponentCount`

The number of elements in the **Components** array at the end of this structure. This count includes all elements of the array, and includes both the first element, which is a member of this structure, and any additional elements that follow the end of this structure.

`ComponentActiveConditionCallback`

A pointer to a ComponentActiveConditionCallback callback routine. PoFx calls this routine to notify the driver that the specified component completed a transition from the idle condition to the active condition.

`ComponentIdleConditionCallback`

A pointer to a ComponentIdleConditionCallback callback routine. PoFx calls this routine to notify the driver that the specified component completed a transition from the active condition to the idle condition.

`ComponentCriticalTransitionCallback`

A pointer to a ComponentCriticalTransitionCallback callback routine. PoFx calls this routine to notify the driver that the specified component is about to make a transition between the idle condition and the active condition.

`ComponentIdleStateCallback`

A pointer to a ComponentIdleStateCallback callback routine. PoFx calls this routine to notify the driver of a pending change to the F*x* power state of the specified component.

`DeviceContext`

A pointer to a PEP-defined device context. PoFx supplies this pointer as the first parameter in calls to the *ComponentActiveConditionCallback*, *ComponentIdleConditionCallback*, *ComponentCriticalTransitionCallback*, and

ComponentIdleStateCallback routines. This context is opaque to PoFx.

`Components`

The first element in an array of PO_FX_COMPONENT structures. Each element in the array describes the power attributes of a component in the device. If the array contains more than one element, the additional elements immediately follow the end of the **PO_FX_CORE_DEVICE** structure. The number of elements in the array is specified in the **ComponentCount** member of the **PO_FX_CORE_DEVICE** structure.

Remarks

This structure is used by the PoFxRegisterCoreDevice routine.

A *core system resource* is a hardware device, such as a timer or an interrupt controller, that is managed by the Windows hardware abstraction layer (HAL).

For more information about component transitions between the active condition and the idle condition, see Component-Level Power Management.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	pepfx.h (include Pep_x.h)

See Also

ComponentActiveConditionCallback

ComponentCriticalTransitionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

PO_FX_COMPONENT

PoFxRegisterCoreDevice

PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.

Syntax

```
void PEP_ACPI_INITIALIZE_EXTENDED_IO_RESOURCE(
  BOOLEAN            ResourceUsage,
  UCHAR              Decode,
  BOOLEAN            IsMinFixed,
  BOOLEAN            IsMaxFixed,
  UCHAR              ISARanges,
  ULONGLONG          AddressGranularity,
  ULONGLONG          AddressMinimum,
  ULONGLONG          AddressMaximum,
  ULONGLONG          AddressTranslation,
  ULONGLONG          RangeLength,
  ULONGLONG          TypeSpecificAttributes,
  PUNICODE_STRING    DescriptorName,
  BOOLEAN            TranslationTypeNonStatic,
  BOOLEAN            TanslationSparseDensity,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ResourceUsage`

This parameter is copied into the **GeneralFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`Decode`

When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only.

When not set, indicates that this bridge positively decodes this address.

`IsMinFixed`

When set, indicates that the minimum address is fixed.

`IsMaxFixed`

When set, indicates that the maximum address is fixed.

`ISARanges`

This parameter is copied into the **TypeSpecificFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`AddressGranularity`

A bit mask indicating which bits have been decoded.

`AddressMinimum`

For bridges that translate addresses, this indicates the minimum starting address on the secondary side of the bridge.

`AddressMaximum`

For bridges that translate addresses, this indicates the maximum starting address on the secondary side of the bridge.

`AddressTranslation`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`RangeLength`

The length of the address range.

`TypeSpecificAttributes`

The type-specific attributes for this resource.

`DescriptorName`

The name of the resource descriptor.

`TranslationTypeNonStatic`

When true, indicates that the resource uses type translation. Otherwise, it uses type-static translation.

`TanslationSparseDensity`

When false, indicates that this is a dense translation. Otherwise, it is sparse.

`Resource`

This is cast to *PEP_ACPI_EXTENDED_ADDRESS.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_EXTENDED_ADDRESS structure.

Syntax

```
void PEP_ACPI_INITIALIZE_EXTENDED_MEMORY_RESOURCE(
  BOOLEAN            ResourceUsage,
  UCHAR              Decode,
  BOOLEAN            IsMinFixed,
  BOOLEAN            IsMaxFixed,
  UCHAR              Cacheable,
  BOOLEAN            ReadWrite,
  ULONGLONG          AddressGranularity,
  ULONGLONG          AddressMinimum,
  ULONGLONG          AddressMaximum,
  ULONGLONG          AddressTranslation,
  ULONGLONG          RangeLength,
  ULONGLONG          TypeSpecificAttributes,
  PUNICODE_STRING    DescriptorName,
  UCHAR              MemoryRangeType,
  BOOLEAN            TranslationTypeNonStatic,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ResourceUsage`

This parameter is copied into the **GeneralFlags** member of the initialized PEP_ACPI_EXTENDED_ADDRESS structure.

`Decode`

When set, indicates that this bridge subtractively decodes the address. This applies to top level bridges only.

When not set, indicates that this bridge positively decodes this address.

`IsMinFixed`

When set, indicates that the minimum address is fixed.

`IsMaxFixed`

When set, indicates that the maximum address is fixed.

`Cacheable`

The caching flag for the resource.

VALUE	MEANING

0	Indicates the memory is non-cacheable.
1	Indicates the memory is cacheable.
2	Indicates the memory is cacheable and supports write combining.
3	The memory is cacheable and prefetchable.

`ReadWrite`

When true, indicates that the resource is available for read/write access. Otherwise, it's read-only.

`AddressGranularity`

A bit mask indicating which bits have been decoded.

`AddressMinimum`

For bridges that translate addresses, this indicates the minimum starting address on the secondary side of the bridge.

`AddressMaximum`

For bridges that translate addresses, this indicates the maximum starting address on the secondary side of the bridge.

`AddressTranslation`

For bridges that translate addresses across the bridge, this is the address on the primary side.

`RangeLength`

The length of the address range.

`TypeSpecificAttributes`

The type-specific attributes for this resource.

`DescriptorName`

The name of the resource descriptor.

`MemoryRangeType`

This parameter identifies the type of memory range provided by this resource.

VALUE	MEANING
0	Address range memory: This range is available RAM usable by the operating system.
1	Address range reserved: This range of addresses is in use or reserved by the system and is not to be included in the allocatable memory pool of the operating system's memory manager.
2	Address range ACPI: ACPI Reclaim Memory. This range is available RAM usable by the OS after it reads the ACPI tables.

3	Address Range NVS: ACPI NVS Memory. This range of addresses is in use or reserved by the system and must not be used by the operating system. This range is required to be saved and restored across an NVS sleep.

`TranslationTypeNonStatic`

When true, indicates that the resource uses type translation. Otherwise, it uses type-static translation.

`Resource`

This is cast to *PEP_ACPI_EXTENDED_ADDRESS.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_EXTENDED_ADDRESS

PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_GPIO_INT_RESOURCE(
  KINTERRUPT_MODE      InterruptType,
  KINTERRUPT_POLARITY  LevelType,
  BOOLEAN              Shareable,
  BOOLEAN              CanWake,
  GPIO_PIN_CONFIG_TYPE PinConfig,
  USHORT               DebounceTimeout,
  UCHAR                ResourceSourceIndex,
  PUNICODE_STRING      ResourceSourceName,
  BOOLEAN              ResourceUsage,
  PUCHAR               VendorData,
  USHORT               VendorDataLength,
  PUSHORT              PinTable,
  UCHAR                PinCount,
  PPEP_ACPI_RESOURCE   Resource
);
```

Parameters

`InterruptType`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`LevelType`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`Shareable`

Indicates if the device can be shared.

`CanWake`

Indicates if the device can be woken from a low-power state.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceSourceName`

This parameter should always be "ResourceConsumer."

`ResourceUsage`

Indicates if this device is in use.

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the buffer in the *VendorData* partameter.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_GPIO_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

KINTERRUPT_MODE

KINTERRUPT_POLARITY

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_GPIO_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_GPIO_IO_RESOURCE(
  BOOLEAN                     Shareable,
  BOOLEAN                     CanWake,
  GPIO_PIN_CONFIG_TYPE        PinConfig,
  USHORT                      DebounceTimeout,
  USHORT                      DriveStrength,
  GPIO_PIN_IORESTRICTION_TYPE IoRestriction,
  UCHAR                       ResourceSourceIndex,
  PUNICODE_STRING             ResourceSourceName,
  BOOLEAN                     ResourceUsage,
  PUCHAR                      VendorData,
  USHORT                      VendorDataLength,
  PUSHORT                     PinTable,
  USHORT                      PinCount,
  PPEP_ACPI_RESOURCE          Resource
);
```

Parameters

`Shareable`

Indicates if the device can be shared.

`CanWake`

Indicates if the device can be woken from a low-power state.

`PinConfig`

A GPIO_PIN_CONFIG_TYPE enumeration value that identifies the GPIO pin configuration type.

`DebounceTimeout`

Specifies the hardware debounce wait time, in hundredths of milliseconds.

`DriveStrength`

`IoRestriction`

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceSourceName`

This parameter should always be "ResourceConsumer."

`ResourceUsage`

Indicates if this device is in use.

`VendorData`

A pointer to a raw data buffer containing vendor-defined byte data to be decoded by the OS driver.

`VendorDataLength`

The size of the buffer in the *VendorData* partameter.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_GPIO_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

GPIO_PIN_CONFIG_TYPE

PEP_ACPI_GPIO_RESOURCE

PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_INTERRUPT_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_INTERRUPT_RESOURCE(
  BOOLEAN             ResourceUsage,
  KINTERRUPT_MODE     EdgeLevel,
  KINTERRUPT_POLARITY InterruptLevel,
  BOOLEAN             ShareType,
  BOOLEAN             Wake,
  PULONG              PinTable,
  UCHAR               PinCount,
  PPEP_ACPI_RESOURCE  Resource
);
```

Parameters

`ResourceUsage`

Indicates if this device is in use.

`EdgeLevel`

A KINTERRUPT_MODE enumeration value that identifies the interrupt type.

`InterruptLevel`

A KINTERRUPT_POLARITY enumeration value that identifies how a device signals an interrupt request on an interrupt line.

`ShareType`

Indicates if the device can be shared.

`Wake`

Indicates if the device can be woken from a low-power state.

`PinTable`

A list of pin numbers on the resource.

`PinCount`

The number of pins described by the *PinTable* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_INTERRUPT_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_INTERRUPT_RESOURCE

PEP_ACPI_INITIALIZE_IOPORT_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_IOPORT_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_IOPORT_RESOURCE(
  UCHAR               Decode,
  USHORT              MinimumAddress,
  USHORT              MaximumAddress,
  UCHAR               Alignment,
  UCHAR               PortLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`Decode`

Specifies how to decode the resource address. If bit 0 is a 1, this indicates that the logical device decodes 16-bit addresses. If bit 0 is 0, this indicates that the logical device only decodes the first 10 bits of the address.

Bits 1 to 7 of this parameter are reserved and must be set to zero.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the IO address assigned.

`PortLength`

Specifies the number of bytes in the IO range.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_IO_MEMORY_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_INITIALIZE_MEMORY_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_MEMORY_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_IO_MEMORY_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_MEMORY_RESOURCE(
  UCHAR              ReadWrite,
  ULONG              MinimumAddress,
  ULONG              MaximumAddress,
  ULONG              Alignment,
  ULONG              MemorySize,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`ReadWrite`

If true, indicates that the resource is read/write. Otherwise, it's read-only.

`MinimumAddress`

Specifies the minimum acceptable starting address for the IO range.

`MaximumAddress`

Specifies the maximum acceptable starting address for the IO range.

`Alignment`

Specifies the alignment granularity for the memory address assigned.

`MemorySize`

Specifies the number of bytes in the memory range.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_IO_MEMORY_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.

Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_IO_MEMORY_RESOURCE

PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_I2C_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_I2C_RESOURCE(
  USHORT              SlaveAddress,
  BOOLEAN             DeviceInitiated,
  ULONG               ConnectionSpeed,
  BOOLEAN             AddressingMode,
  PUNICODE_STRING     ResourceSource,
  UCHAR               ResourceSourceIndex,
  BOOLEAN             ResourceUsage,
  BOOLEAN             SharedMode,
  PCHAR               VendorData,
  USHORT              VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`SlaveAddress`

The I2C bus address for this connection.

`DeviceInitiated`

If true, indicates that communication over this connection is initiated by the device.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`AddressingMode`

Indicates that this device is in addressing mode.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be set to zero.

`ResourceUsage`

Indicates if the resource is in use.

`SharedMode`

Indicates if the resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_I2C_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_I2C_RESOURCE

PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_SPI_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_SPI_RESOURCE(
  USHORT              DeviceSelection,
  UCHAR               DeviceSelectionPolarity,
  UCHAR               WireMode,
  UCHAR               DataBitLength,
  BOOLEAN             SlaveMode,
  ULONG               ConnectionSpeed,
  UCHAR               ClockPolarity,
  UCHAR               ClockPhase,
  PUNICODE_STRING     ResourceSource,
  UCHAR               ResourceSourceIndex,
  BOOLEAN             ResourceUsage,
  BOOLEAN             SharedMode,
  PCHAR               VendorData,
  USHORT              VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`DeviceSelection`

The device selection value. This value is specific to the device and may refer to a chip-select line, GPIO line, or other line selection mechanism.

`DeviceSelectionPolarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`WireMode`

When zero, indicates that this device produces and consumes this resource. Otherwise, this device only consumes this resource.

`DataBitLength`

The size, in bits, of the smallest unit of transfer.

`SlaveMode`

Indicates if the resource is operating in slave mode.

`ConnectionSpeed`

The maximum speed, in hertz, supported by this connection.

`ClockPolarity`

The polarity of the clock. If zero, this indicates the clock is low during the first phase. If 1, this indicates the clock is high during the first phase.

`ClockPhase`

The phase of the clock pulse on which to capture data.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be set to zero.

`ResourceUsage`

Indicates if the resource is in use.

`SharedMode`

Indicates if the resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_SPI_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_SPI_RESOURCE

PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE function

1/25/2019 • 2 minutes to read • Edit Online

The **PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE** function initializes a platform extension plug-in's (PEP) PEP_ACPI_SPB_UART_RESOURCE structure.

Syntax

```
void PEP_ACPI_INITIALIZE_SPB_UART_RESOURCE(
  ULONG              BaudRate,
  UCHAR              BitsPerByte,
  UCHAR              StopBits,
  UCHAR              LinesInUse,
  UCHAR              IsBigEndian,
  UCHAR              Parity,
  UCHAR              FlowControl,
  USHORT             RxSize,
  USHORT             TxSize,
  PUNICODE_STRING    ResourceSource,
  UCHAR              ResourceSourceIndex,
  BOOLEAN            ResourceUsage,
  BOOLEAN            SharedMode,
  PCHAR              VendorData,
  USHORT             VendorDataLength,
  PPEP_ACPI_RESOURCE Resource
);
```

Parameters

`BaudRate`

Specifies the baud rate of the connection.

`BitsPerByte`

Specifies the number of bits per byte of data.

`StopBits`

Specifies the stop bits used in the connection.

`LinesInUse`

Flag indicating the serial lines that are enabled. A value of 1 in the bit positions indicates that the line is enabled.

BIT	MEANING
0	This bit is reserved and must be set to zero.
1	This bit is reserved and must be set to zero.

2	Data Carrier Detect (DTD)
3	Ring Indicator (RI)
4	Data Set Ready (DSR)
5	Data Terminal Ready (DTR)
6	Clear to Send (CTS)
7	Request to Send (RTS)

`IsBigEndian`

Indicates if the most significant bits of data are in the lowest address.

`Parity`

Specifies the parity of the connection.

VALUE	MEANING
0x0 0	None
0x0 1	Even
0x0 2	Odd
0x0 3	Mark
0x0 4	Space

`FlowControl`

Specifies the type of flow control used by the connection.

`RxSize`

Specifies the maximum receive buffer size, in bytes, that is supported by this connection.

`TxSize`

Specifies the maximum transmit buffer size, in bytes, that is supported by this connection.

`ResourceSource`

The name of the serial bus controller device to which this connection descriptor applies. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules.

`ResourceSourceIndex`

This parameter should always be zero.

`ResourceUsage`

Indicates if this resource is in use.

`SharedMode`

Indicates if this resource is shared.

`VendorData`

A pointer to optional data that is specific to the serial bus connection type.

`VendorDataLength`

The length of the buffer pointed to by the *VendorData* parameter.

`Resource`

A pointer to the resource. The structure behind the pointer is of type PEP_ACPI_SPB_UART_RESOURCE.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_ACPI_SPB_UART_RESOURCE

PEPCALLBACKNOTIFYACPI callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptAcpiNotification* event callback routine handles ACPI notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYACPI Pepcallbacknotifyacpi;

BOOLEAN Pepcallbacknotifyacpi(
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Notification`

The ACPI notification ID. This parameter specifies the type of ACPI notification that PoFx is sending. For a list of ACPI notification IDs, see ACPI notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various ACPI notification IDs, see ACPI notifications.

Return Value

The *AcceptAcpiNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a device. The **AcceptAcpiNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptAcpiNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptAcpiNotification* routine with PoFx.

The IRQL at which the PEP's *AcceptAcpiNotification* routine is called varies according to the type of notification that is being sent. The notification type is specified by the *Notification* parameter. For more information about the IRQL at which each notification type is sent, see the individual reference pages under ACPI notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.

Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKNOTIFYDPM callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptDeviceNotification* event callback routine handles device power management (DPM) notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYDPM Pepcallbacknotifydpm;

BOOLEAN Pepcallbacknotifydpm(
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Notification`

The DPM notification ID. This parameter specifies the type of DPM notification that PoFx is sending. For a list of DPM notification IDs, see Device power management (DPM) notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various DPM notification IDs, see Device power management (DPM) notifications.

Return Value

The *AcceptDeviceNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a device. The **AcceptDeviceNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptDeviceNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptDeviceNotification* routine with PoFx.

PoFx calls this routine to send a DPM notification to the PEP. In this call, the *Notification* parameter specifies the type of notification, and the *Data* parameter points to a data structure of the type that is associated with this notification. For a list of DPM notification IDs and their associated structure types, see Device power management (DPM) notifications.

DPM notifications inform the PEP about various system, device, or componentÂ–-level events, or pass requests from the device driver to the PEP.

The *AcceptDeviceNotification* routine does not have to accept every request from the device driver—for example, it can ignore idle residency hints. However, the PEP must understand and accept most device power management notifications even if the PEP chooses not to act on them. The PEP must refuse to handle (by returning FALSE) every notification that it does not recognize.

The *AcceptDeviceNotification* routine can be called at IRQL <= DISPATCH_LEVEL. The IRQL at which the PEP's *AcceptAcpiNotification* routine is called varies according to the type of notification that is being sent. The notification type is specified by the *Notification* parameter. For more information, see the individual reference pages under Device power management (DPM) notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKNOTIFYPPM callback function

1/25/2019 • 2 minutes to read • Edit Online

An *AcceptProcessorNotification* event callback routine handles processor power management (PPM) notifications from the Windows power management framework (PoFx).

Syntax

```
PEPCALLBACKNOTIFYPPM Pepcallbacknotifyppm;

BOOLEAN Pepcallbacknotifyppm(
  PEPHANDLE Handle,
  ULONG Notification,
  PVOID Data
)
{...}
```

Parameters

`Handle`

The device handle for the PEP for the target processor. If **NULL**, the notification does not target a specific processor.

`Notification`

The PPM notification ID. This parameter specifies the type of PPM notification that PoFx is sending. For a list of PPM notification IDs, see Processor power management (PPM) notifications.

`Data`

A pointer to a PoFx-allocated structure that contains the input and/or output data for this notification. The type of this structure depends on the notification ID specified by *Notification*. For a list of the structure types that are defined for the various PPM notification IDs, see Processor power management (PPM) notifications.

Return Value

The *AcceptProcessorNotification* routine must return TRUE if it handles the notification, or FALSE if it does not.

Remarks

This routine is implemented by the platform extension plug-in (PEP) for a processor. The **AcceptProcessorNotification** member of the PEP_INFORMATION structure is a pointer to an *AcceptProcessorNotification* routine. The PEP calls the PoFxRegisterPlugin or PoFxRegisterPluginEx routine to register the PEP's *AcceptProcessorNotification* routine with PoFx.

The IRQL at which the PEP's *AcceptProcessorNotification* routine is called varies according to the type of notification that is being sent. The context in which this interface is called ranges from PASSIVE_LEVEL to HIGH_LEVEL with interrupts disabled. For more information about the IRQL at which each notification type is sent, see the individual reference pages under Processor power management (PPM) notifications.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL to HIGH_LEVEL

See Also

PEP_INFORMATION

PoFxRegisterPlugin

PoFxRegisterPluginEx

PEPCALLBACKPOWERONCRASHDUMPDEVICE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *PowerOnDumpDeviceCallback* callback routine turns on the crash-dump device.

Syntax

```
PEPCALLBACKPOWERONCRASHDUMPDEVICE Pepcallbackpoweroncrashdumpdevice;

BOOLEAN Pepcallbackpoweroncrashdumpdevice(
  PPEP_CRASHDUMP_INFORMATION CrashdumpInformation
)
{...}
```

Parameters

`CrashdumpInformation`

A pointer to a PEP_CRASHDUMP_INFORMATION structure that contains information about the crash-dump device.

Return Value

This routine returns TRUE if it succeeds in turning on power to the crash-dump device. Otherwise, it returns FALSE.

Remarks

This routine is implemented by the platform extension plug-in (PEP). The routine handles requests from the Windows kernel to turn on the crash-dump device so that a crash dump can be saved to disk.

The PEP registers its *PowerOnDumpDeviceCallback* routine in response to a PEP_DPM_REGISTER_CRASHDUMP_DEVICE notification from the Windows power management framework (PoFx).

The *PowerOnDumpDeviceCallback* routine is called at IRQL = HIGH_LEVEL with interrupts disabled. Thus, this routine cannot use timers, events, or any synchronization primitives other than spin locks.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

IRQL	Called at HIGH_LEVEL.

See Also

PEP_CRASHDUMP_INFORMATION

PEP_DPM_REGISTER_CRASHDUMP_DEVICE

POFXCALLBACKCRITICALRESOURCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **TransitionCriticalResource** routine changes the active/inactive state of a core system component.

Syntax

```
POFXCALLBACKCRITICALRESOURCE Pofxcallbackcriticalresource;

void Pofxcallbackcriticalresource(
  POHANDLE DeviceHandle,
  ULONG Component,
  BOOLEAN Active
)
{...}
```

Parameters

`DeviceHandle`

A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

`Component`

[in] The index that identifies the component. This member is an index into the **Components** array in the PEP_DEVICE_REGISTER_V2 structure that the PEP previously supplied in response to the **PEP_DPM_REGISTER_DEVICE** notification for this device. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`Active`

Whether to transition the device from inactive to active, or vice versa. Set to TRUE to transition the device from inactive to active, or to FALSE to transition the device from active to inactive.

Return Value

None

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **TransitionCriticalResource** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **TransitionCriticalResource** routine.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_DEVICE_REGISTER_V2

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS callback function

1/25/2019 • 2 minutes to read • Edit Online

The **EnumerateUnmaskedInterrupts** routine enumerates interrupt sources whose interrupts are unmasked and enabled.

Syntax

```
POFXCALLBACKENUMERATEUNMASKEDINTERRUPTS Pofxcallbackenumerateunmaskedinterrupts;

NTSTATUS Pofxcallbackenumerateunmaskedinterrupts(
  POHANDLE PluginHandle,
  ULONG EnumerateFlags,
  PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK Callback,
  PVOID CallbackContext,
  PPEP_UNMASKED_INTERRUPT_INFORMATION InterruptInformation
)
{...}
```

Parameters

`PluginHandle`

A POHANDLE value. If non-NULL, this parameter is a handle that identifies the platform extension plug-in (PEP), in which case **EnumerateUnmaskedInterrupts** enumerates only interrupts that are managed by this PEP. If this parameter is NULL, **EnumerateUnmaskedInterrupts** enumerates *all* interrupts in the hardware platform that are unmasked and enabled.

`EnumerateFlags`

No flags are currently defined. Set this parameter to PEP_ENUMERATE_UNMASKED_INTERRUPT_FLAGS_NONE (0x0).

`Callback`

A pointer to a caller-implemented EnumerateInterruptSource callback routine. This callback routine is called once for each interrupt source whose interrupt is unmasked. These callbacks occur synchronously before the **EnumerateUnmaskedInterrupts** routine returns.

`CallbackContext`

A pointer to a callback context. This pointer is passed as a parameter to the *EnumerateInterruptSource* callback routine pointed to by the *Callback* parameter. The contents of the callback context are PEP-defined, and are opaque to the power management framework (PoFx).

`InterruptInformation`

A pointer to a caller-allocated buffer whose size is at least **sizeof**(PEP_UNMASKED_INTERRUPT_INFORMATION) bytes. **EnumerateUnmaskedInterrupts** will use this buffer to transfer interrupt information to the PEP during calls to the PEP's *EnumerateInterruptSource* callback routine.

Return Value

EnumerateUnmaskedInterrupts returns STATUS_SUCCESS if the call successfully enumerates the interrupts. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The **Size** or **Version** member of the **PEP_UNMASKED_INTERRUPT_INFORMATION** structure contains an invalid value.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **EnumerateUnmaskedInterrupts** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **EnumerateUnmaskedInterrupts** routine.

Before the platform enters a system power state in which the interrupt controllers are power-gated, the PEP can call **EnumerateUnmaskedInterrupts** to get the information it needs to properly configure wake-up controllers for interrupts that are to remain unmasked.

The PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

EnumerateInterruptSource

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_UNMASKED_INTERRUPT_INFORMATION

POFXCALLBACKPLATFORMIDLEVETO callback function

1/25/2019 • 2 minutes to read • Edit Online

The **PlatformIdleVeto** routine increments or decrements the veto count for a veto code for a platform idle state.

Syntax

```
POFXCALLBACKPLATFORMIDLEVETO Pofxcallbackplatformidleveto;

NTSTATUS Pofxcallbackplatformidleveto(
  POHANDLE ProcessorHandle,
  ULONG PlatformState,
  ULONG VetoReason,
  BOOLEAN Increment
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with power management framework (PoFx). The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`PlatformState`

The index of the platform idle state whose veto count is being incremented or decremented. The PEP previously specified the supported platform idle states in response to a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification. If the PEP specified N platform idle states, valid platform idle state indexes range from 0 to N–1.

`VetoReason`

A PEP-defined veto code that indicates why the platform idle state cannot be entered. The PEP previously specified the number of supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. If the PEP specified M veto reasons, valid veto codes range from 1 to M.

`Increment`

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Return Value

PlatformIdleVeto returns STATUS_SUCCESS if the call successfully increments or decrements the veto count. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_I MPL EME NTE D	Indicates that the **PlatformIdleVeto** routine is not implemented for this processor.

Remarks

This routine is implemented by the PoFx and is called by the PEP. The **PlatformIdleVeto** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **PlatformIdleVeto** routine.

After the PEP handles the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification, each platform idle state starts with a veto count of zero for all veto codes. The PEP can increment the veto count of a veto code to prevent the operating system from transitioning the platform to this idle state. The PEP decrements the veto count when the veto reason that caused the count to be incremented is no longer in effect. Only after the count for all veto codes has decremented to zero is the operating system allowed to transition the platform to this idle state.

This routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

POFXCALLBACKPROCESSORHALT callback function

1/25/2019 • 3 minutes to read • Edit Online

The **ProcessorHalt** routine prepares the processor to be halted.

Syntax

```
POFXCALLBACKPROCESSORHALT Pofxcallbackprocessorhalt;

NTSTATUS Pofxcallbackprocessorhalt(
  ULONG Flags,
  PVOID Context,
  PPROCESSOR_HALT_ROUTINE Halt
)
{...}
```

Parameters

`Flags`

Flags that indicate the properties of the idle state that the processor will enter. The *Flags* parameter is set to zero or to the bitwise-OR of one or more of the following flag bits.

FLAG NAME	VALUE	DESCRIPTION
PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE	0x01	The PEP's *Halt* callback routine is responsible for flushing the processor's caches. If this flag is not set, **ProcessorHalt** will flush the caches on entry to the processor idle state, and will invalidate the caches on exit from this state.
PROCESSOR_HALT_CACHE_COHERENT	0x02	Prepare the processor to enter an idle state in which the processor's caches will remain coherent.
PROCESSOR_HALT_CONTEXT_RETAINED	0x04	Prepare the processor to enter an idle state in which the processor's thread context will be preserved.
PROCESSOR_HALT_RETURN_NOT_SAFE	0x08	The PEP's *Halt* callback routine is guaranteed to not return. Set this flag if the *Halt* callback transitions the processor to a context-losing, power-gated state that cannot be canceled by hardware. When this flag is set, the operating system will treat a return from the *Halt* callback as a fatal error.
PROCESSOR_HALT_VIA_PSCI_CPU_SUSPEND	0x16	

`Context`

A pointer to a PEP-defined processor-halt context. This pointer is passed as a parameter to the *Halt* callback routine. This context is opaque to the Windows power management framework (PoFx).

`Halt`

A pointer to a PEP-implemented *Halt* callback routine. PoFx calls this routine after preparations to halt the processor have been completed. During this callback, the PEP is expected to transition the processor to the *halted* state.

Return Value

ProcessorHalt returns STATUS_SUCCESS if the processor is successfully prepared to be halted. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The *Halt* parameter is NULL; or an invalid flag value was specified in *Flags*; or *Flags* contains an illegal combination of flag bits. For more information, see Remarks.
STA TUS _UN SUC CES SFU L	The PEP's *Halt* callback routine unexpectedly returned from an idle state in which the processor's hardware context was not preserved.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **ProcessorHalt** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **ProcessorHalt** routine.

Before halting the processor, the PEP calls the **ProcessorHalt** routine to give PoFx an opportunity to save the processor's hardware context. If necessary, **ProcessorHalt** saves this state internally in PoFx so that the state can later be restored when the processor exits the idle state. After preparing the processor to enter the idle state, **ProcessorHalt** calls the PEP's *Halt* callback routine to halt the processor.

As part of the PEP's handling of a PEP_NOTIFY_PPM_IDLE_EXECUTE notification, the PEP must transition the processor to the idle state that the PEP has selected. The following are the two ways to enter the processor idle state:

- For a processor idle state in which the processor's caches remain coherent so that all system and processor state is maintained, the PEP can enter the idle state directly without first calling **ProcessorHalt**.
- For a processor idle state in which the processor's caches might not remain coherent, or an idle state in which the processor hardware context is not preserved, the PEP must call **ProcessorHalt** before transitioning the processor to the idle state.

The following combinations of flag bits are illegal:

- PROCESSOR_HALT_CONTEXT_RETAINED = 1 and PROCESSOR_HALT_RETURN_NOT_SAFE = 1

  The PEP's *Halt* callback routine must return from any state in which context is retained.

- PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE = 1 and PROCESSOR_HALT_CACHE_COHERENT = 1

  The cache-flush-override flag should be set only if entering an idle state that is not cache coherent.

- PROCESSOR_HALT_CACHE_FLUSH_OVERRIDE = 0 and PROCESSOR_HALT_CACHE_COHERENT = 0

  The cache-flush-override flag must be set for any non-cache coherent halt.

- PROCESSOR_HALT_CONTEXT_RETAINED = 0 and PROCESSOR_HALT_CACHE_COHERENT = 1

  Any idle states that lose processor hardware context (and therefore use the multiprocessor parking protocol to exit the idle state and return control to the operating system) are not cache-coherent states.

If the *Flags* parameter contains an illegal combination of flag bits, **ProcessorHalt** fails and returns STATUS_INVALID_PARAMETER.

The PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_IDLE_EXECUTE

POFXCALLBACKPROCESSORIDLEVETO callback function

1/25/2019 • 2 minutes to read • Edit Online

The **ProcessorIdleVeto** routine increments or decrements the veto count for a veto code for a processor idle state.

Syntax

```
POFXCALLBACKPROCESSORIDLEVETO Pofxcallbackprocessoridleveto;

NTSTATUS Pofxcallbackprocessoridleveto(
  POHANDLE ProcessorHandle,
  ULONG ProcessorState,
  ULONG VetoReason,
  BOOLEAN Increment
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with power management framework (PoFx). The platform extension plug-in (PEP) previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`ProcessorState`

The index of the processor idle state whose veto count is being incremented or decremented. The PEP previously specified the supported processor idle states in response to a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification. If the PEP specified N processor idle states, valid processor idle state indexes range from 0 to N–1.

`VetoReason`

A PEP-defined veto code that indicates why the processor idle state cannot be entered. The PEP previously specified the supported veto reasons in response to a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification. If the PEP specified M veto reasons, valid veto codes range from 1 to M.

`Increment`

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Whether to increment or decrement the veto count. Set to TRUE to increment the veto count for this veto reason, or to FALSE to decrement the count.

Return Value

This routine does not return a value.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **ProcessorIdleVeto** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **ProcessorIdleVeto** routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTCOMMON callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestCommon** routine is a generic request handler.

Syntax

```
POFXCALLBACKREQUESTCOMMON Pofxcallbackrequestcommon;

NTSTATUS Pofxcallbackrequestcommon(
  ULONG RequestId,
  PVOID Data
)
{...}
```

Parameters

`RequestId`

A request ID that specifies the operation being requested.

`Data`

A pointer to a data structure that contains the input data and/or result data for the request specified by the *RequestId* parameter.

Return Value

RequestCommon returns STATUS_SUCCESS if the request is successfully handled. Otherwise, it returns an appropriate error status code.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **RequestCommon** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **RequestCommon** routine.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)

IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTINTERRUPT callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestInterrupt** routine requests that the operating system replay an edge-triggered interrupt that might have been lost while the hardware platform was in a low-power state.

Syntax

```
POFXCALLBACKREQUESTINTERRUPT Pofxcallbackrequestinterrupt;

NTSTATUS Pofxcallbackrequestinterrupt(
  ULONG Gsiv
)
{...}
```

Parameters

`Gsiv`

The global system interrupt vector (GSIV) number that identifies this interrupt. The ACPI firmware assigns GSIV numbers to all primary interrupt lines. For secondary (GPIO) interrupt lines, the GSIV number is dynamically assigned by the operating system.

Return Value

RequestInterrupt returns STATUS_SUCCESS if the specified interrupt is successfully replayed. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	Not a valid GSIV number.

Remarks

This routine is implemented by the power management framework (PoFx) and is called by the platform extension plug-in (PEP). The **RequestInterrupt** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **RequestInterrupt** routine.

Assertion of an edge-triggered interrupt signal is transient in nature and can be lost if the power to a primary interrupt controller is gated off when the interrupt occurs. In contrast, a level-triggered interrupt signal is expected

to remain asserted until it can be serviced.

A PEP can call this routine at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= HIGH_LEVEL"

See Also

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKREQUESTWORKER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **RequestWorker** routine is called by a platform extension plug-in (PEP) to inform the Windows power management framework (PoFx) that the platform extension plug-in (PEP) has a work request to submit on behalf of the specified device.

Syntax

```
POFXCALLBACKREQUESTWORKER Pofxcallbackrequestworker;

void Pofxcallbackrequestworker(
  POHANDLE PluginHandle
)
{...}
```

Parameters

`PluginHandle`

A POHANDLE value that represents the registration of the device with PoFx. The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the device's driver stack registered the device with PoFx.

Return Value

None.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **RequestWorker** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to a **RequestWorker** routine.

Each time the PEP calls this routine, PoFx responds by sending a PEP_DPM_WORK notification to the PEP. This notification might be delayed if a worker thread is not immediately available to process the work request.

A PEP can call this routine at IRQL <= DISPATCH_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

PEP_DPM_WORK

PEP_KERNEL_INFORMATION_STRUCT_V3

POFXCALLBACKUPDATEPLATFORMIDLESTATE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **UpdatePlatformIdleState** routine is called by the platform extension plug-in (PEP) to update the properties of the specified platform idle state.

Syntax

```
POFXCALLBACKUPDATEPLATFORMIDLESTATE Pofxcallbackupdateplatformidlestate;

NTSTATUS Pofxcallbackupdateplatformidlestate(
  POHANDLE ProcessorHandle,
  ULONG PlatformState,
  PPEP_PLATFORM_IDLE_STATE_UPDATE Update
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`PlatformState`

An index that identifies the platform idle state whose properties are to be updated. In response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification, the PEP specified the number of supported platform idle states. If the PEP specified N platform idle states, valid platform idle state indexes range from 0 to N-1. In response to a previous PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification, the PEP specified the properties of this platform idle state.

`Update`

A pointer to a PEP_PLATFORM_IDLE_STATE_UPDATE structure that contains the updated properties of the platform idle state.

Return Value

UpdatePlatformIdleState returns STATUS_SUCCESS if it successfully updates the properties of the platform idle state. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_S UPP ORT ED	The version number in the **PEP_PLATFORM_IDLE_STATE_UPDATE** structure is not a supported value.
STA TUS _NO T_I MPL EME NTE D	The **UpdatePlatformIdleState** routine is not implemented for this processor.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **UpdatePlatformIdleState** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **UpdatePlatformIdleState** routine.

The PEP must wait until after it has completed all PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notifications to call the **UpdatePlatformIdleState** routine.

The **UpdatePlatformIdleState** routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

PEP_PLATFORM_IDLE_STATE_UPDATE

POFXCALLBACKUPDATEPROCESSORIDLESTATE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **UpdateProcessorIdleState** routine is called by the platform extension plug-in (PEP) to update the properties of the specified processor idle state.

Syntax

```
POFXCALLBACKUPDATEPROCESSORIDLESTATE Pofxcallbackupdateprocessoridlestate;

NTSTATUS Pofxcallbackupdateprocessoridlestate(
  POHANDLE ProcessorHandle,
  ULONG ProcessorState,
  PPEP_PROCESSOR_IDLE_STATE_UPDATE Update
)
{...}
```

Parameters

`ProcessorHandle`

A POHANDLE value that represents the registration of the processor (as a device) with the Windows power management framework (PoFx). The PEP previously received this handle from PoFx during the PEP_DPM_REGISTER_DEVICE notification that informed the PEP that the processor had been registered with PoFx.

`ProcessorState`

An index that identifies the processor idle state whose properties are to be updated. In response to a previous PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification, the PEP specified the number of supported processor idle states and the properties of these states. If the PEP specified N processor idle states, valid processor idle state indexes range from 0 to N-1.

`Update`

A pointer to a PEP_PROCESSOR_IDLE_STATE_UPDATE structure that contains the updated properties of the processor idle state.

Return Value

UpdateProcessorIdleState returns STATUS_SUCCESS if it successfully updates the properties of the processor idle state. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION

STA TUS _NO T_S UPP ORT ED	The version number in the **PEP_PROCESSOR_IDLE_STATE_UPDATE** structure is not a supported value.
STA TUS _NO T_I MPL EME NTE D	The **UpdateProcessorIdleState** routine is not implemented for this processor.

Remarks

This routine is implemented by PoFx and is called by the PEP. The **UpdateProcessorIdleState** member of the PEP_KERNEL_INFORMATION_STRUCT_V3 structure is a pointer to an **UpdateProcessorIdleState** routine.

The **UpdateProcessorIdleState** routine must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	PASSIVE_LEVEL

See Also

PEP_DPM_REGISTER_DEVICE

PEP_KERNEL_INFORMATION_STRUCT_V3

PEP_PROCESSOR_IDLE_STATE_UPDATE

PoFxRegisterCoreDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterCoreDevice** routine registers a new core system resource with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterCoreDevice(
  PCUNICODE_STRING   Id,
  PPO_FX_CORE_DEVICE Device,
  POHANDLE           *Handle
);
```

Parameters

`Id`

A string that uniquely identifies the core system resource. This member is a pointer to a UNICODE_STRING structure that contains a device identification string.

`Device`

A pointer to a PO_FX_CORE_DEVICE structure that describes the power characteristics of the core system resource.

`Handle`

A pointer to a location to which the routine writes a POHANDLE value. This handle represents the registration of the core system resource with PoFx.

Return Value

PoFxRegisterCoreDevice returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The component count for this device is zero, or the **PO_FX_CORE_DEVICE** structure contains an invalid version number, or the **ComponentCriticalTransitionCallback** member of this structure contains an invalid function pointer.

STA TUS _INS UFFI CIE NT_ RES OU RCE S	Unable to allocate the resources required to complete the requested registration.

Remarks

This routine registers a core system resource so that this resource can be power-managed by PoFx.

Core system resources are hardware devices, such as timers and interrupt controllers, that are managed by the Windows hardware abstraction layer (HAL). These devices provide basic functions that are required by the operating system. Due to the close relationship between core system resources and processors, the power management of these resources needs to be coordinated with processor idle state management.

The **PoFxRegisterCoreDevice** routine should be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PO_FX_CORE_DEVICE

UNICODE_STRING

PoFxRegisterPlugin function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterPlugin** routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterPlugin(
  PPEP_INFORMATION        PepInformation,
  PPEP_KERNEL_INFORMATION KernelInformation
);
```

Parameters

`PepInformation`

A pointer to a PEP_INFORMATION structure.

`KernelInformation`

A pointer to a PEP_KERNEL_INFORMATION_STRUCT_V3 structure.

Return Value

PoFxRegisterPlugin returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STATUS_INVALID_PARAMETER	The **Version** or **Size** member of the **PEP_KERNEL_INFORMATION** structure is set to an invalid value; or the **AcceptDeviceNotification** member of this structure is set to NULL.
STATUS_INVALID_PEP_INFO_VERSION	The **Version** member of the **PEP_INFORMATION** structure is set to an invalid value.
STATUS_INSUFFICIENT_RESOURCES	Unable to allocate the resources required to complete the requested registration.

Remarks

A PEP calls this routine to register itself with PoFx.

A PEP cannot unregister, and cannot register twice. If the PEP must be serviced, the operating system must restart.

The PoFxRegisterPluginEx routine is similar to **PoFxRegisterPlugin**, except that it takes an additional parameter, *Flags*.

The PEP must call **PoFxRegisterPlugin** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PEP_INFORMATION

PEP_KERNEL_INFORMATION_STRUCT_V3

PoFxRegisterPluginEx

PoFxRegisterPluginEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterPluginEx** routine registers a platform extension plug-in (PEP) with the Windows power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterPluginEx(
  PPEP_INFORMATION        PepInformation,
  ULONGLONG               Flags,
  PPEP_KERNEL_INFORMATION KernelInformation
);
```

Parameters

`PepInformation`

A pointer to a PEP_INFORMATION structure that contains pointers to one or more callback routines that are implemented by the PEP. These routines handle notifications that are sent to the PEP by PoFx.

`Flags`

A set of flag bits for configuring the PEP interface. Set this member to zero or to the following value.

FLAG BIT	DESCRIPTION
PEP_FLAG_WORKER_CONCURRENCY	

`KernelInformation`

A pointer to a PEP_KERNEL_INFORMATION structure.

Return Value

PoFxRegisterPluginEx returns STATUS_SUCCESS if the call successfully registers the PEP. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	The **Version** or **Size** member of the **PEP_KERNEL_INFORMATION** structure is set to an invalid value; or the **AcceptDeviceNotification** member of this structure is set to NULL.

STA TUS _IN VAL ID_P EP_I NFO _VE RSI ON	The **Version** member of the **PEP_INFORMATION** structure is set to an invalid value.
STA TUS _INS UFFI CIE NT_ RES OU RCE S	Unable to allocate the resources required to complete the requested registration.

Remarks

A PEP calls this routine to register itself with PoFx.

A PEP cannot unregister, and cannot register twice. If the PEP must be serviced, the operating system must restart.

The PoFxRegisterPlugin routine is similar to **PoFxRegisterPluginEx**, except that it does not take a *Flags* parameter.

The PEP must call **PoFxRegisterPluginEx** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PEP_INFORMATION

PEP_KERNEL_INFORMATION

PoFxRegisterPlugin

PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

An *EnumerateInterruptSource* callback routine supplies a platform extension plug-in (PEP) with information about an interrupt source.

Syntax

```
PPO_ENUMERATE_INTERRUPT_SOURCE_CALLBACK PpoEnumerateInterruptSourceCallback;

BOOLEAN PpoEnumerateInterruptSourceCallback(
  PVOID CallbackContext,
  PPEP_UNMASKED_INTERRUPT_INFORMATION InterruptInformation
)
{...}
```

Parameters

`CallbackContext`

A pointer to a callback context. The PEP specified this pointer value as a parameter in the EnumerateUnmaskedInterrupts call that initiated the enumeration of interrupt sources.

`InterruptInformation`

A pointer to a PEP_UNMASKED_INTERRUPT_INFORMATION structure that contains information about the interrupt source.

Return Value

If the *EnumerateInterruptSource* callback routine returns TRUE, **EnumerateUnmaskedInterrupts** will continue to call the *EnumerateInterruptSource* callback routine while more interrupts are available to be enumerated. If the *EnumerateInterruptSource* callback routine returns FALSE, **EnumerateUnmaskedInterrupts** returns without enumerating any more interrupts.

Remarks

This callback routine is implemented by a PEP, and is called by the Windows power management framework (PoFx). The *Callback* parameter of the EnumerateUnmaskedInterrupts routine is a pointer to an *EnumerateInterruptSource* callback routine.

After a PEP calls **EnumerateUnmaskedInterrupts** to enumerate the unmasked interrupt sources, PoFx calls the PEP's *EnumerateInterruptSource* callback routine once for each unmasked interrupt source. **EnumerateUnmaskedInterrupts** returns only after the last call to the *EnumerateInterruptSource* callback routine completes.

An *EnumerateInterruptSource* callback routine is called at the same IRQL as the PEP's call to **EnumerateUnmaskedInterrupts** that initiates the enumeration callbacks.

Requirements

Target Platform	Windows
Header	pepfx.h (include Pep_x.h)
IRQL	See Remarks.

See Also

PEP_UNMASKED_INTERRUPT_INFORMATION

EnumerateUnmaskedInterrupts

procgrp.h header

1/8/2019 • 2 minutes to read • Edit Online

This provides the new kernel programming interfaces for device drivers to run on downlevel systems. The library implements wrapper functions for processor-group **Ke*Xxx*** routines.

- Windows kernel procgrp.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
WdmlibProcgrpInitialize	The WdmlibProcgrpInitialize function initializes the Processor Group (ProcGrp) compatibility library.

WdmlibProcgrpInitialize function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibProcgrpInitialize** function initializes the Processor Group (ProcGrp) compatibility library.

Syntax

```
void WdmlibProcgrpInitialize(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

This function initializes the ProcGrp library. Call this function before calling any of the other functions in the ProcGrp library. The library implements wrapper functions that have the same names as the following processor-group **Ke*Xxx*** routines in Windows 7:

KeGetCurrentProcessorNumberEx

KeGetProcessorIndexFromNumber

KeGetProcessorNumberFromIndex

KeQueryActiveGroupCount

KeQueryActiveProcessorCountEx

KeQueryGroupAffinity

KeQueryMaximumProcessorCount

KeQueryMaximumProcessorCountEx

KeQueryMaximumGroupCount

KeSetSystemAffinityThreadEx

KeSetSystemGroupAffinityThread

KeRevertToUserAffinityThreadEx

KeRevertToUserGroupAffinityThread

KeSetTargetProcessorDpcEx

For more information about the ProcGrp library, see Processor Group Compatibility Library.

Requirements

Minimum supported client	Compatible with Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, and Windows 2000.
Target Platform	Desktop
Header	procgrp.h (include Procgrp.h)
Library	Procgrp.lib
IRQL	PASSIVE_LEVEL

vpci.h header

1/8/2019 • 2 minutes to read • Edit Online

This header contains common types and prototypes for a virtual PCI driver.

- vpci.h contains the following programming interfaces:

IOCTLs

TITLE	DESCRIPTION
IOCTL_VPCI_INVALIDATE_BLOCK	The driver for a PCI Express (PCIe) virtual function (VF) issues the IOCTL_VPCI_INVALIDATE_BLOCK IOCTL request in order to be notified of changes to data in one or more VF configuration blocks.
IOCTL_VPCI_READ_BLOCK	The driver for a PCI Express (PCIe) virtual function (VF) issues an IOCTL_VPCI_READ_BLOCK I/O control code (IOCTL) in order to read data from a VF configuration block. The driver issues this IOCTL to the next-lower driver in the driver stack.
IOCTL_VPCI_WRITE_BLOCK	The driver for a PCI Express (PCIe) virtual function (VF) issues an IOCTL_VPCI_WRITE_BLOCK I/O control code (IOCTL) in order to write data to a VF configuration block. The driver issues this IOCTL to the next-lower driver in the driver stack.

Structures

TITLE	DESCRIPTION
VPCI_INVALIDATE_BLOCK_OUTPUT	The VPCI_INVALIDATE_BLOCK_OUTPUT structure is used in an IOCTL_VPCI_INVALIDATE_BLOCK IOCTL request.
VPCI_READ_BLOCK_INPUT	The VPCI_READ_BLOCK_INPUT structure is used in an IOCTL_VPCI_READ_BLOCK IOCTL request to read data from a specified configuration block of data for a PCI Express (PCIe) virtual function (VF).
VPCI_WRITE_BLOCK_INPUT	The VPCI_WRITE_BLOCK_INPUT structure is used in an IOCTL_VPCI_WRITE_BLOCK IOCTL request to write data to a specified configuration block for a PCI Express (PCIe) virtual function (VF).

wdm.h header

1/8/2019 • 109 minutes to read • Edit Online

This header contains reference material that includes specific details about the routines, structures, and data types that you will need to use to write kernel-mode drivers.

For more information, see:

- Windows kernel wdm.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
AppendTailList	The AppendTailList routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of LIST_ENTRY structures.
ARM64_SYSREG_CRM	Returns the CRn field for a given ARM system register.
ARM64_SYSREG_CRN	Returns the CRn field for a given ARM system register.
ARM64_SYSREG_OP1	Returns the op1 field for a given ARM system register.
ARM64_SYSREG_OP2	Returns the op2 field for a given ARM system register.
ASSERTMSG	ASSERTMSG tests an expression. If the expression is false, it breaks into the kernel debugger and sends it the specified message.
ClfsAddLogContainer	The ClfsAddLogContainer routine adds a container to a CLFS log.
ClfsAddLogContainerSet	The ClfsAddLogContainerSet routine atomically adds a set of containers to a CLFS log.
ClfsAdvanceLogBase	The ClfsAdvanceLogBase routine sets the base LSN of a CLFS stream.
ClfsAlignReservedLog	The ClfsAlignReservedLog routine calculates the size of the space that must be reserved for a specified set of records. The size calculation includes the space required for headers and the space required for sector alignment.
ClfsAllocReservedLog	The ClfsAllocReservedLog routine reserves space in a marshalling area for a set of records.
ClfsCloseAndResetLogFile	The ClfsCloseAndResetLogFile routine releases all references to a specified log file object and marks its associated stream for reset.

TITLE	DESCRIPTION
ClfsCloseLogFileObject	The ClfsCloseLogFileObject routine releases all references to a log file object.
ClfsCreateLogFile	The ClfsCreateLogFile routine creates or opens a CLFS stream. If necessary, ClfsCreateLogFile also creates the underlying physical log that holds the stream's records.
ClfsCreateMarshallingArea	The ClfsCreateMarshallingArea routine creates a marshalling area for a CLFS stream and returns a pointer to an opaque context that represents the new marshalling area.
ClfsCreateMarshallingAreaEx	Initalizes a marshalling area for a physical or client log file stream.
ClfsCreateScanContext	The ClfsCreateScanContext routine creates a scan context that can be used to iterate over the containers of a specified CLFS log.
ClfsDeleteLogByPointer	The ClfsDeleteLogByPointer routine marks a CLFS stream for deletion.
ClfsDeleteLogFile	The ClfsDeleteLogFile routine marks a CLFS stream for deletion.
ClfsDeleteMarshallingArea	The ClfsDeleteMarshallingArea routine deletes a marshalling area.
ClfsFlushBuffers	The ClfsFlushBuffers routine forces all log I/O blocks in a specified marshalling area to stable storage.
ClfsFlushToLsn	The ClfsFlushToLsn routine forces, to stable storage, all records that have an LSN less than or equal to a specified LSN.
ClfsGetContainerName	The ClfsGetContainerName routine returns the path name of a specified container.
ClfsGetIoStatistics	The ClfsGetIoStatistics routine returns I/O statistics for a specified CLFS log.
ClfsLsnBlockOffset	The ClfsLsnBlockOffset routine returns the sector-aligned block offset contained in a specified LSN.
ClfsLsnContainer	The ClfsLsnContainer routine returns the logical container identifier contained in a specified LSN.
ClfsLsnCreate	The ClfsLsnCreate routine creates a log sequence number (LSN), given a container identifier, a block offset, and a record sequence number.
ClfsLsnEqual	The ClfsLsnEqual routine determines whether two LSNs from the same stream are equal.

TITLE	DESCRIPTION
ClfsLsnGreater	The ClfsLsnGreater routine determines whether one LSN is greater than another LSN. The two LSNs must be from the same stream.
ClfsLsnLess	The ClfsLsnLess routine determines whether one LSN is less than another LSN. The two LSNs must be from the same stream.
ClfsLsnNull	The ClfsLsnNull routine determines whether a specified LSN is equal to the smallest possible LSN, CLFS_LSN_NULL.
ClfsLsnRecordSequence	The ClfsLsnRecordSequence routine returns the record sequence number contained in a specified LSN.
ClfsMgmtDeregisterManagedClient	The ClfsMgmtDeregisterManagedClient routine removes the connection between a client and a log, so that the client no longer manages the log.
ClfsMgmtHandleLogFileFull	The ClfsMgmtHandleLogFileFull routine attempts to make more space available in a log. It might make more space available by adding containers to the log, or it might ask clients to move their log tails.
ClfsMgmtInstallPolicy	The ClfsMgmtInstallPolicy routine adds a CLFS_MGMT_POLICY structure to a physical log.
ClfsMgmtQueryPolicy	The ClfsMgmtQueryPolicy routine retrieves a specific CLFS_MGMT_POLICY structure for a log.
ClfsMgmtRegisterManagedClient	The ClfsMgmtRegisterManagedClient routine creates a client that will manage a CLFS log.
ClfsMgmtRemovePolicy	The ClfsMgmtRemovePolicy routine resets a log's CLFS_MGMT_POLICY structure to its default value.
ClfsMgmtSetLogFileSize	The ClfsMgmtSetLogFileSize routine adds containers to a log or deletes containers from a log.
ClfsMgmtSetLogFileSizeAsClient	The ClfsMgmtSetLogFileSizeAsClient routine sets the log file size by adding containers to a client log or deleting containers from a client log.
ClfsMgmtTailAdvanceFailure	The ClfsMgmtTailAdvanceFailure routine notifies CLFS management that the client could not advance the log's tail.
ClfsQueryLogFileInformation	The ClfsQueryLogFileInformation routine returns metadata and state information for a specified CLFS stream or its underlying physical log or both.
ClfsReadLogRecord	The ClfsReadLogRecord routine reads a target record in a CLFS stream and returns a read context that the caller can use to read the records preceding or following it in the stream.
ClfsReadNextLogRecord	The ClfsReadNextLogRecord routine reads the next record in a sequence, relative to the current record in a read context.

TITLE	DESCRIPTION
ClfsReadPreviousRestartArea	The ClfsReadPreviousRestartArea routine reads the previous restart record relative to the current record in a read context.
ClfsReadRestartArea	The ClfsReadRestartArea routine reads the restart record that was most recently written to a specified CLFS stream.
ClfsRemoveLogContainer	The ClfsRemoveLogContainer routine removes a container from a CLFS log.
ClfsRemoveLogContainerSet	The ClfsRemoveLogContainerSet routine atomically removes a set of containers from a CLFS log.
ClfsReserveAndAppendLog	The ClfsReserveAndAppendLog routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically.
ClfsReserveAndAppendLogAligned	The ClfsReserveAndAppendLogAligned routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically. The record's data is aligned on specified boundaries.
ClfsScanLogContainers	The ClfsScanLogContainers routine retrieves descriptive information for a sequence of containers that belong to a particular CLFS log.
ClfsSetArchiveTail	The ClfsSetArchiveTail routine sets the archive tail of a CLFS log to a specified LSN.
ClfsSetEndOfLog	The ClfsSetEndOfLog routine truncates a CLFS stream.
ClfsSetLogFileInformation	The ClfsSetLogFileInformation routine sets metadata and state information for a specified stream and its underlying physical log.
ClfsTerminateReadLog	The ClfsTerminateReadLog routine invalidates a specified read context after freeing resources associated with the context.
ClfsWriteRestartArea	The ClfsWriteRestartArea routine atomically appends a new restart record to a CLFS stream, flushes the restart record to stable storage, and optionally updates the base LSN of the stream.
CmCallbackGetKeyObjectID	The CmCallbackGetKeyObjectID routine retrieves the unique identifier and object name that are associated with a specified registry key object.
CmCallbackGetKeyObjectIDEx	The CmCallbackGetKeyObjectIDEx routine retrieves the unique identifier and object name that are associated with a specified registry key object.

TITLE	DESCRIPTION
CmCallbackReleaseKeyObjectIDEx	The CmCallbackReleaseKeyObjectIDEx routine frees an object name string obtained from the CmCallbackGetKeyObjectIDEx routine.
CmGetBoundTransaction	The CmGetBoundTransaction routine returns a pointer to the transaction object that represents the transaction, if any, that is associated with a specified registry key object.
CmGetCallbackVersion	The CmGetCallbackVersion routine retrieves the major and minor version numbers for the current version of the configuration manager's registry callback feature.
CmRegisterCallback	The CmRegisterCallback routine is obsolete for Windows Vista and later operating system versions. Use CmRegisterCallbackEx instead.The CmRegisterCallback routine registers a RegistryCallback routine.
CmRegisterCallbackEx	The CmRegisterCallbackEx routine registers a RegistryCallback routine.
CmSetCallbackObjectContext	The CmSetCallbackObjectContext routine associates specified context information with a specified registry object.
CmUnRegisterCallback	The CmUnRegisterCallback routine unregisters a RegistryCallback routine that a CmRegisterCallback or CmRegisterCallbackEx routine previously registered.
DbgBreakPoint	The DbgBreakPoint routine breaks into the kernel debugger.
DbgBreakPointWithStatus	The DbgBreakPointWithStatus routine breaks into the kernel debugger and sends the value of Status to the debugger.
DbgPrint	The DbgPrint routine sends a message to the kernel debugger.
DbgPrintEx	The DbgPrintEx routine sends a string to the kernel debugger if the conditions you specify are met.
EtwActivityIdControl	The EtwActivityIdControl function creates, queries, and sets the current activity identifier.
EtwEventEnabled	The EtwEventEnabled function verifies whether an event is enabled.
EtwProviderEnabled	The EtwProviderEnabled function verifies that a provider is enabled for event logging at a specified level and keyword.
EtwRegister	The EtwRegister function registers the event provider and must be called before a provider can start tracing.
EtwUnregister	The EtwUnregister function unregisters the event provider and must be called before the provider exits.
EtwWrite	The EtwWrite function is a tracing function for publishing events in your kernel-mode driver code.

TITLE	DESCRIPTION
EtwWriteEx	The EtwWriteEx function is a tracing function for publishing events that support filtering in your kernel-mode driver code.
EtwWriteString	The EventWriteString function is a tracing function that you can use when no sophisticated data is required. This function is similar to a debug print statement.
EtwWriteTransfer	The EtwWriteTransfer function marks an event that links two activities together; this type of event is referred to as a transfer event.
ExAcquirePushLockExclusive	Acquires the given push lock for exclusive access by the calling thread.
ExAcquirePushLockShared	Acquires the given push lock for shared access by the calling thread.
ExAcquireResourceExclusiveLite	The ExAcquireResourceExclusiveLite routine acquires the given resource for exclusive access by the calling thread.
ExAcquireResourceSharedLite	The ExAcquireResourceSharedLite routine acquires the given resource for shared access by the calling thread.
ExAcquireRundownProtection	The ExAcquireRundownProtection routine tries to acquire run-down protection on a shared object so the caller can safely access the object.
ExAcquireRundownProtectionEx	The ExAcquireRundownProtectionEx routine tries to acquire run-down protection on a shared object so the caller can safely access the object.
ExAcquireSharedStarveExclusive	The ExAcquireSharedStarveExclusive routine acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.
ExAcquireSharedWaitForExclusive	The ExAcquireSharedWaitForExclusive routine acquires the given resource for shared access if shared access can be granted and there are no exclusive waiters.
ExAllocateFromLookasideListEx	The ExAllocateFromLookasideListEx routine removes the first entry from the specified lookaside list, or, if the list is empty, dynamically allocates the storage for a new entry.
ExAllocateFromNPagedLookasideList	The ExAllocateFromNPagedLookasideList routine returns a pointer to a nonpaged entry from the given lookaside list, or it returns a pointer to a newly allocated nonpaged entry.
ExAllocateFromPagedLookasideList	The ExAllocateFromPagedLookasideList routine returns a pointer to a paged entry from the given lookaside list, or it returns a pointer to a newly allocated paged entry.
ExAllocatePool	The ExAllocatePool routine is obsolete, and is exported only for existing binaries. Use ExAllocatePoolWithTag instead.ExAllocatePool allocates pool memory of the specified type and returns a pointer to the allocated block.

TITLE	DESCRIPTION
ExAllocatePoolWithQuota	The ExAllocatePoolWithQuota routine is obsolete, and is exported only for existing driver binaries. Use ExAllocatePoolWithQuotaTag instead.ExAllocatePoolWithQuota allocates pool memory, charging quota against the current process.
ExAllocatePoolWithQuotaTag	The ExAllocatePoolWithQuotaTag routine allocates pool memory, charging the quota against the current process.
ExAllocatePoolWithTag	The ExAllocatePoolWithTag routine allocates pool memory of the specified type and returns a pointer to the allocated block.
ExAllocatePoolWithTagPriority	The ExAllocatePoolWithTagPriority routine allocates pool memory of the specified type.
ExAllocateTimer	The ExAllocateTimer routine allocates and initializes a timer object.
ExCancelTimer	The ExCancelTimer routine cancels a timer that was set by a previous call to the ExSetTimer routine.
ExConvertExclusiveToSharedLite	The ExConvertExclusiveToSharedLite routine converts a given resource from acquired for exclusive access to acquired for shared access.
ExCreateCallback	The ExCreateCallback routine either creates a new callback object or opens an existing callback object on behalf of the caller.
ExDeleteLookasideListEx	The ExDeleteLookasideListEx routine deletes a lookaside list.
ExDeleteNPagedLookasideList	The ExDeleteNPagedLookasideList routine destroys a nonpaged lookaside list.
ExDeletePagedLookasideList	The ExDeletePagedLookasideList routine destroys a paged lookaside list.
ExDeleteResourceLite	The ExDeleteResourceLite routine deletes a given resource from the system's resource list.
ExDeleteTimer	The ExDeleteTimer routine deletes a timer object that was previously allocated by the ExAllocateTimer routine.
ExFlushLookasideListEx	The ExFlushLookasideListEx routine flushes all entries from the specified lookaside list and frees the allocated storage for each entry.
ExFreePool	The ExFreePool routine deallocates a block of pool memory.
ExFreePoolWithTag	The ExFreePoolWithTag routine deallocates a block of pool memory allocated with the specified tag.

TITLE	DESCRIPTION
ExFreeToLookasideListEx	The ExFreeToLookasideListEx routine inserts an entry into a lookaside list, or, if the list is full, frees the allocated storage for the entry.
ExFreeToNPagedLookasideList	The ExFreeToNPagedLookasideList routine returns a nonpaged entry to the given lookaside list or to nonpaged pool.
ExFreeToPagedLookasideList	The ExFreeToPagedLookasideList routine returns a pageable entry to the given lookaside list or to paged pool.
ExGetExclusiveWaiterCount	The ExGetExclusiveWaiterCount routine returns the number of waiters on exclusive access to a given resource.
ExGetFirmwareEnvironmentVariable	The ExGetFirmwareEnvironmentVariable routine gets the value of the specified system firmware environment variable.
ExGetFirmwareType	Returns the system firmware type.
ExGetPreviousMode	The ExGetPreviousMode routine returns the previous processor mode for the current thread.
ExGetSharedWaiterCount	The ExGetSharedWaiterCount routine returns the number of waiters on shared access to a given resource.
ExInitializeDeleteTimerParameters	The ExInitializeDeleteTimerParameters routine initializes an EXT_DELETE_PARAMETERS structure.
ExInitializeFastMutex	The ExInitializeFastMutex routine initializes a fast mutex variable, used to synchronize mutually exclusive access by a set of threads to a shared resource.
ExInitializeLookasideListEx	The ExInitializeLookasideListEx routine initializes a lookaside list.
ExInitializeNPagedLookasideList	The ExInitializeNPagedLookasideList routine initializes a lookaside list for nonpaged entries of the specified size.
ExInitializePagedLookasideList	The ExInitializePagedLookasideList routine initializes a lookaside list for pageable entries of the specified size.
ExInitializePushLock	Initializes a push lock variable.
ExInitializeResourceLite	The ExInitializeResourceLite routine initializes a resource variable.
ExInitializeRundownProtection	The ExInitializeRundownProtection routine initializes run-down protection on a shared object.
ExInitializeSetTimerParameters	The ExInitializeSetTimerParameters routine initializes an EXT_SET_PARAMETERS structure.
ExInitializeWorkItem	ExInitializeWorkItem initializes a work-queue item with a caller-supplied context and callback routine to be queued for execution when a system worker thread is given control.

TITLE	DESCRIPTION
ExInterlockedAddLargeInteger	The ExInterlockedAddLargeInteger routine adds a large integer value to the specified variable as an atomic operation.
ExInterlockedAddLargeStatistic	The ExInterlockedAddLargeStatistic routine performs an interlocked addition of a ULONG increment value to a LARGE_INTEGER variable.
ExInterlockedAddUlong	The ExInterlockedAddUlong routine adds an unsigned long value to a given unsigned integer as an atomic operation.
ExInterlockedCompareExchange64	The ExInterlockedCompareExchange64 routine compares one integer variable to another and, if they are equal, sets the first variable to a caller-supplied value.
ExInterlockedFlushSList	The ExInterlockedFlushSList routine atomically removes all entries from a sequenced singly linked list.
ExInterlockedInsertHeadList	The ExInterlockedInsertHeadList routine atomically inserts an entry at the beginning of a doubly linked list of LIST_ENTRY structures.
ExInterlockedInsertTailList	The ExInterlockedInsertTailList routine atomically inserts an entry at the end of a doubly linked list of LIST_ENTRY structures.
ExInterlockedPopEntryList	The ExInterlockedPopEntryList routine atomically removes an entry from the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
ExInterlockedPopEntrySList	The ExInterlockedPopEntrySList routine atomically removes the first entry from a sequenced singly linked list.
ExInterlockedPushEntryList	The ExInterlockedPushEntryList routine atomically inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
ExInterlockedPushEntrySList	The ExInterlockedPushEntrySList routine atomically inserts an entry at the beginning of a sequenced singly linked list.
ExInterlockedRemoveHeadList	The ExInterlockedRemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
ExIsProcessorFeaturePresent	The ExIsProcessorFeaturePresent routine queries for the existence of a specified processor feature.
ExIsResourceAcquiredExclusiveLite	The ExIsResourceAcquiredExclusiveLite routine returns whether the current thread has exclusive access to a given resource.
ExIsResourceAcquiredSharedLite	The ExIsResourceAcquiredSharedLite routine returns whether the current thread has access (either shared or exclusive) to a given resource.

TITLE	DESCRIPTION
ExIsSoftBoot	Determines whether the system has gone through a soft restart.
ExLocalTimeToSystemTime	The ExLocalTimeToSystemTime routine converts a system time value for the current time zone to an unbiased, GreenGMT value.
ExNotifyCallback	The ExNotifyCallback routine causes all callback routines registered for the given object to be called.
ExQueryDepthSList	The ExQueryDepthSList routine returns the number of entries currently in a given sequenced singly linked list.
ExQueryTimerResolution	The ExQueryTimerResolution routine reports the range of timer resolutions that are supported by the system clock.
ExQueueWorkItem	ExQueueWorkItem inserts a given work item into a queue from which a system worker thread removes the item and gives control to the routine that the caller supplied to ExInitializeWorkItem.
ExRaiseStatus	The ExRaiseStatus routine is called by drivers that supply structured exception handlers to handle particular errors that occur while they are processing I/O requests.
ExRegisterCallback	The ExRegisterCallback routine registers a given callback routine with a given callback object.
ExReinitializeResourceLite	The ExReinitializeResourceLite routine reinitializes an existing resource variable.
ExReInitializeRundownProtection	The ExReInitializeRundownProtection routine reinitializes an EX_RUNDOWN_REF structure after the associated object is run down.
ExReleasePushLockExclusive	Releases a specified push lock for exclusive access owned by the current thread.
ExReleasePushLockShared	Releases a specified push lock for shared access owned by the current thread.
ExReleaseResourceForThreadLite	The ExReleaseResourceForThreadLite routine releases the input resource of the indicated thread.
ExReleaseResourceLite	The ExReleaseResourceLite routine releases a specified executive resource owned by the current thread.
ExReleaseRundownProtection	The ExReleaseRundownProtection routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtection routine.
ExReleaseRundownProtectionEx	The ExReleaseRundownProtectionEx routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtectionEx routine.

TITLE	DESCRIPTION
ExReleaseSpinLockExclusive	The ExReleaseSpinLockExclusive routine releases a spin lock that the caller previously acquired for exclusive access, and restores the IRQL to its original value.
ExReleaseSpinLockShared	The ExReleaseSpinLockShared routine releases ownership of a spin lock that the caller previously acquired for shared access, and restores the IRQL to its original value.
ExRundownCompleted	The ExRundownCompleted routine updates the run-down status of a shared object to indicate that the run down of the object has completed.
ExSetFirmwareEnvironmentVariable	The ExSetFirmwareEnvironmentVariable routine sets the value of the specified system firmware environment variable.
ExSetResourceOwnerPointer	The ExSetResourceOwnerPointer routine sets the owner thread pointer for an executive resource.
ExSetResourceOwnerPointerEx	The ExSetResourceOwnerPointerEx routine transfers the ownership of an executive resource from the calling thread to an owner pointer, which is a system address that identifies the resource owner.
ExSetTimer	The ExSetTimer routine starts a timer operation and sets the timer to expire at the specified due time.
ExSetTimerResolution	The ExSetTimerResolution routine modifies the frequency at which the system clock interrupts. Use this routine with extreme caution (see the following Remarks section).
ExSystemTimeToLocalTime	The ExSystemTimeToLocalTime routine converts a GMT system time value to the local system time for the current time zone.
ExTryConvertSharedSpinLockExclusive	The ExTryConvertSharedSpinLockExclusive routine attempts to convert the access state of a spin lock from acquired for shared access to exclusive access.
ExUnregisterCallback	The ExUnregisterCallback routine removes a callback routine previously registered with a callback object from the list of routines to be called during the notification process.
ExWaitForRundownProtectionRelease	The ExWaitForRundownProtectionRelease routine waits until all drivers that have already been granted run-down protection complete their accesses of the shared object.
FIELD_OFFSET	The FIELD_OFFSET macro returns the byte offset of a named field in a known structure type.
FirstEntrySList	The FirstEntrySList routine returns the first entry in a sequenced singly linked list.
IMAGE_POLICY_OVERRIDE	Reserved.

TITLE	DESCRIPTION
InitializeListHead	The InitializeListHead routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.
InitializeSListHead	The InitializeSListHead routine (or ExInitializeSListHead) initializes an SLIST_HEADER structure that represents the head of a sequenced singly linked list.
InsertHeadList	The InsertHeadList routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.
InsertTailList	The InsertTailList routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.
InterlockedAnd	The InterlockedAnd macro atomically computes a bitwise AND operation.
InterlockedCompareExchange	The InterlockedCompareExchange routine performs an atomic operation that compares the input value pointed to by Destination with the value of Comperand.
InterlockedCompareExchangePointer	The InterlockedCompareExchangePointer routine performs an atomic operation that compares the input pointer value pointed to by Destination with the pointer value Comperand.
InterlockedDecrement	The InterlockedDecrement routine decrements a caller-supplied variable of type LONG as an atomic operation.
InterlockedExchange	The InterlockedExchange routine sets an integer variable to a given value as an atomic operation.
InterlockedExchangeAdd	The InterlockedExchangeAdd routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.
InterlockedExchangePointer	The InterlockedExchangePointer routine performs an atomic operation that sets a pointer to a new value.
InterlockedIncrement	The InterlockedIncrement routine increments a caller-supplied variable as an atomic operation.
InterlockedOr	The InterlockedOr routine atomically computes a bitwise OR operation.
InterlockedXor	The InterlockedOr routine atomically computes a bitwise exclusive OR operation.
IoAcquireRemoveLock	The IoAcquireRemoveLock routine increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack or deleted.
IoAdjustPagingPathCount	The IoAdjustPagingPathCount routine increments or decrements a caller-supplied page-file counter as an atomic operation.

TITLE	DESCRIPTION
IoAllocateDriverObjectExtension	The IoAllocateDriverObjectExtension routine allocates a per-driver context area, called a driver object extension, and assigns a unique identifier to it.
IoAllocateErrorLogEntry	The IoAllocateErrorLogEntry routine allocates an error log entry, and returns a pointer to the packet that the caller uses to supply information about an I/O error.
IoAllocateIrp	The IoAllocateIrp routine allocates an IRP, given the number of I/O stack locations for each driver layered under the caller, and, optionally, for the caller.
IoAllocateMdl	The IoAllocateMdl routine allocates a memory descriptor list (MDL) large enough to map a buffer, given the buffer's starting address and length. Optionally, this routine associates the MDL with an IRP.
IoAllocateWorkItem	The IoAllocateWorkItem routine allocates a work item.
IoAttachDevice	The IoAttachDevice routine attaches the caller's device object to a named target device object, so that I/O requests bound for the target device are routed first to the caller.
IoAttachDeviceToDeviceStack	The IoAttachDeviceToDeviceStack routine attaches the caller's device object to the highest device object in the chain and returns a pointer to the previously highest device object.
IoBuildAsynchronousFsdRequest	The IoBuildAsynchronousFsdRequest routine allocates and sets up an IRP to be sent to lower-level drivers.
IoBuildDeviceIoControlRequest	The IoBuildDeviceIoControlRequest routine allocates and sets up an IRP for a synchronously processed device control request.
IoBuildPartialMdl	The IoBuildPartialMdl routine builds a new memory descriptor list (MDL) that represents part of a buffer that is described by an existing MDL.
IoBuildSynchronousFsdRequest	The IoBuildSynchronousFsdRequest routine allocates and sets up an IRP for a synchronously processed I/O request.
IoCallDriver	The IoCallDriver routine sends an IRP to the driver associated with a specified device object.
IoCancelIrp	The IoCancelIrp routine sets the cancel bit in a given IRP and calls the cancel routine for the IRP if there is one.
IoCheckLinkShareAccess	The IoCheckLinkShareAccess routine is called by file system drivers (FSDs) or other highest-level drivers to check whether link shared access to a file object is permitted.
IoCheckShareAccess	The IoCheckShareAccess routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.

TITLE	DESCRIPTION
IoCheckShareAccessEx	The IoCheckShareAccessEx routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.
IoCompleteRequest	The IoCompleteRequest routine indicates that the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O manager.
IoConnectInterrupt	The IoConnectInterrupt routine registers a device driver's InterruptService routine (ISR), so that it will be called when a device interrupts on any of a specified set of processors.
IoConnectInterruptEx	For more information, see the WdmlibIoConnectInterruptEx function.#define IoConnectInterruptEx WdmlibIoConnectInterruptEx
IoCopyCurrentIrpStackLocationToNext	The IoCopyCurrentIrpStackLocationToNext routine copies the IRP stack parameters from the current I/O stack location to the stack location of the next-lower driver.
IoCreateDevice	The IoCreateDevice routine creates a device object for use by a driver.
IoCreateFile	The IoCreateFile routine either causes a new file or directory to be created, or it opens an existing file, device, directory, or volume, giving the caller a handle for the file object.
IoCreateNotificationEvent	The IoCreateNotificationEvent routine creates or opens a named notification event used to notify one or more threads of execution that an event has occurred.
IoCreateSymbolicLink	The IoCreateSymbolicLink routine sets up a symbolic link between a device object name and a user-visible name for the device.
IoCreateSynchronizationEvent	The IoCreateSynchronizationEvent routine creates or opens a named synchronization event for use in serialization of access to hardware between two otherwise unrelated drivers.
IoCreateSystemThread	The IoCreateSystemThread routine creates a system thread that executes in kernel mode, and supplies a handle for the thread.
IoCreateUnprotectedSymbolicLink	The IoCreateUnprotectedSymbolicLink routine sets up an unprotected symbolic link between a device object name and a corresponding Win32-visible name.
IoCsqInitialize	The IoCsqInitialize routine initializes the driver's cancel-safe IRP queue dispatch table.
IoCsqInitializeEx	The IoCsqInitializeEx routine initializes the dispatch table for a cancel-safe IRP queue.
IoCsqInsertIrp	The IoCsqInsertIrp routine inserts an IRP in the driver's cancel-safe IRP queue.

TITLE	DESCRIPTION
IoCsqInsertIrpEx	The IoCsqInsertIrpEx routine inserts an IRP into the driver's cancel-safe IRP queue.
IoCsqRemoveIrp	The IoCsqRemoveIrp routine removes a particular IRP from the queue.
IoCsqRemoveNextIrp	The IoCsqRemoveNextIrp routine removes the next matching IRP in the queue.
IoDeleteDevice	The IoDeleteDevice routine removes a device object from the system, for example, when the underlying device is removed from the system.
IoDeleteSymbolicLink	The IoDeleteSymbolicLink routine removes a symbolic link from the system.
IoDetachDevice	The IoDetachDevice routine releases an attachment between the caller's device object and a lower driver's device object.
IoDisconnectInterrupt	The IoDisconnectInterrupt routine releases a device driver's set of interrupt object(s) when the device is paused or removed, or when the driver is being unloaded.
IoDisconnectInterruptEx	For more information, see the WdmlibIoDisconnectInterruptEx function.#define IoDisconnectInterruptEx WdmlibIoDisconnectInterruptEx
IofCallDriver	Call **IoCallDriver** instead. Sends an IRP to the driver associated with a specified device object.
IoForwardIrpSynchronously	The IoForwardIrpSynchronously routine sends an IRP to a specified driver and waits for that driver to complete the IRP.
IoFreeErrorLogEntry	The IoFreeErrorLogEntry routine frees an unused error log entry.
IoFreeIrp	The IoFreeIrp routine releases a caller-allocated IRP from the caller's IoCompletion routine.
IoFreeMdl	The IoFreeMdl routine releases a caller-allocated memory descriptor list (MDL).
IoFreeWorkItem	The IoFreeWorkItem routine frees a work item that was allocated by IoAllocateWorkItem.
IoGetAffinityInterrupt	For more information, see the WdmlibIoGetAffinityInterrupt function.#define IoGetAffinityInterrupt WdmlibIoGetAffinityInterrupt
IoGetAttachedDeviceReference	The IoGetAttachedDeviceReference routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.

TITLE	DESCRIPTION
IoGetBootDiskInformation	The IoGetBootDiskInformation routine returns information describing the boot and system disks.
IoGetContainerInformation	The IoGetContainerInformation routine provides information about the current state of a user session.
IoGetCurrentIrpStackLocation	The IoGetCurrentIrpStackLocation routine returns a pointer to the caller's I/O stack location in the specified IRP.
IoGetCurrentProcess	The IoGetCurrentProcess routine returns a pointer to the current process.
IoGetDeviceDirectory	Returns a handle to a directory on disk specific to the specified driver object where the driver can read and write files.
IoGetDeviceInterfaceAlias	The IoGetDeviceInterfaceAlias routine returns the alias device interface of the specified device interface instance, if the alias exists.
IoGetDeviceInterfacePropertyData	The IoGetDeviceInterfacePropertyData routine retrieves the current value of a device interface property.
IoGetDeviceInterfaces	The IoGetDeviceInterfaces routine returns a list of device interface instances of a particular device interface class (such as all devices on the system that support a HID interface).
IoGetDeviceNumaNode	The IoGetDeviceNumaNode routine gets the node number of a device.
IoGetDeviceObjectPointer	The IoGetDeviceObjectPointer routine returns a pointer to the top object in the named device object's stack and a pointer to the corresponding file object, if the requested access to the objects can be granted.
IoGetDeviceProperty	The IoGetDeviceProperty routine retrieves information about a device such as configuration information and the name of its PDO.
IoGetDevicePropertyData	The IoGetDevicePropertyData routine retrieves the current setting for a device property.
IoGetDmaAdapter	The IoGetDmaAdapter routine returns a pointer to the DMA adapter structure for a physical device object.
IoGetDriverDirectory	Returns a handle to a directory on disk from which the driver can read and write files. The files in that directory apply to a specific driver object.
IoGetDriverObjectExtension	The IoGetDriverObjectExtension routine retrieves a previously allocated per-driver context area.
IoGetFunctionCodeFromCtlCode	The IoGetFunctionCodeFromCtlCode macro returns the value of the function code contained in an I/O control code.

TITLE	DESCRIPTION
IoGetInitialStack	The IoGetInitialStack routine returns the base address of the current thread's stack.
IoGetIommuInterface	Retrieves a pointer to the interface that contains pointers to IOMMU routines.
IoGetIoPriorityHint	The IoGetIoPriorityHint routine gets the priority hint value from an IRP.
IoGetNextIrpStackLocation	The IoGetNextIrpStackLocation routine gives a higher level driver access to the next-lower driver's I/O stack location in an IRP so the caller can set it up for the lower driver.
IoGetRelatedDeviceObject	Given a file object, the IoGetRelatedDeviceObject routine returns a pointer to the corresponding device object.
IoGetRemainingStackSize	The IoGetRemainingStackSize routine returns the current amount of available kernel-mode stack space.
IoGetStackLimits	The IoGetStackLimits routine returns the boundaries of the current thread's stack frame.
IoGetTopLevelIrp	The IoGetTopLevelIrp routine returns the value of the TopLevelIrp field of the current thread.
IoInitializeDpcRequest	The IoInitializeDpcRequest routine registers a driver-supplied DpcForIsr routine.
IoInitializeIrp	The IoInitializeIrp routine initializes a given IRP that was allocated by the caller.
IoInitializeRemoveLock	The IoInitializeRemoveLock routine initializes a remove lock for a device object.
IoInitializeTimer	The IoInitializeTimer routine sets up a driver-supplied IoTimer routine associated with a given device object.
IoInitializeWorkItem	The IoInitializeWorkItem routine initializes a work item that the caller has already allocated.
IoInvalidateDeviceRelations	The IoInvalidateDeviceRelations routine notifies the PnP manager that the relations for a device (such as bus relations, ejection relations, removal relations, and the target device relation) have changed.
IoInvalidateDeviceState	The IoInvalidateDeviceState routine notifies the PnP manager that some aspect of the PnP state of a device has changed.
IoIs32bitProcess	The IoIs32bitProcess routine checks whether the originator of the current I/O request is a 32-bit user-mode application.
IoIsErrorUserInduced	The IoIsErrorUserInduced routine determines whether an I/O error encountered while processing a request to a removable-media device was caused by the user.

TITLE	DESCRIPTION
IoIsWdmVersionAvailable	The IoIsWdmVersionAvailable routine checks whether a given WDM version is supported by the operating system.
IoMarkIrpPending	The IoMarkIrpPending routine marks the specified IRP, indicating that a driver's dispatch routine subsequently returned STATUS_PENDING because further processing is required by other driver routines.
IoOpenDeviceInterfaceRegistryKey	The IoOpenDeviceInterfaceRegistryKey routine returns a handle to a registry key for storing information about a particular device interface instance.
IoOpenDeviceRegistryKey	The IoOpenDeviceRegistryKey routine returns a handle to a device-specific or a driver-specific registry key for a particular device instance.
IoOpenDriverRegistryKey	Reserved.
IoQueueWorkItem	The IoQueueWorkItem routine associates a WorkItem routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.
IoQueueWorkItemEx	The IoQueueWorkItemEx routine associates a WorkItemEx routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.
IoRegisterContainerNotification	The IoRegisterContainerNotification routine registers a kernel-mode driver to receive notifications about a specified class of events.
IoRegisterDeviceInterface	The IoRegisterDeviceInterface routine registers a device interface class, if it has not been previously registered, and creates a new instance of the interface class, which a driver can subsequently enable for use by applications or other system components.
IoRegisterLastChanceShutdownNotification	The IoRegisterLastChanceShutdownNotification routine registers a driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down, after all file systems have been flushed.
IoRegisterPlugPlayNotification	The IoRegisterPlugPlayNotification routine registers a Plug and Play (PnP) notification callback routine to be called when a PnP event of the specified category occurs.
IoRegisterShutdownNotification	The IoRegisterShutdownNotification routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down.
IoReleaseRemoveLock	The IoReleaseRemoveLock routine releases a remove lock acquired with a previous call to IoAcquireRemoveLock.

TITLE	DESCRIPTION
IoReleaseRemoveLockAndWait	The IoReleaseRemoveLockAndWait routine releases a remove lock that the driver acquired in a previous call to IoAcquireRemoveLock, and waits until all acquisitions of the lock have been released.
IoRemoveLinkShareAccess	The IoRemoveLinkShareAccess routine removes the access and link share-access information for a given open instance of a file object.
IoRemoveShareAccess	The IoRemoveShareAccess routine removes the access and share-access information for a given open instance of a file object.
IoReportInterruptActive	The IoReportInterruptActive routine informs the operating system that a registered interrupt service routine (ISR) is active and ready to handle interrupt requests.
IoReportInterruptInactive	The IoReportInterruptInactive routine informs the operating system that a registered interrupt service routine (ISR) is inactive and is not expecting interrupt requests.
IoReportTargetDeviceChange	The IoReportTargetDeviceChange routine notifies the PnP manager that a custom event has occurred on a device.
IoReportTargetDeviceChangeAsynchronous	The IoReportTargetDeviceChangeAsynchronous routine notifies the PnP manager that a custom event has occurred on a device.
IoRequestDeviceEject	The IoRequestDeviceEject routine notifies the PnP manager that the device eject button was pressed.
IoRequestDpc	The IoRequestDpc routine queues a driver-supplied DpcForIsr routine to complete interrupt-driven I/O processing at a lower IRQL.
IoReuseIrp	The IoReuseIrp routine reinitializes an IRP so that it can be reused.
IoSetCancelRoutine	The IoSetCancelRoutine routine sets up a driver-supplied Cancel routine to be called if a given IRP is canceled.
IoSetCompletionRoutine	The IoSetCompletionRoutine routine registers an IoCompletion routine, which will be called when the next-lower-level driver has completed the requested operation for the given IRP.
IoSetCompletionRoutineEx	The IoSetCompletionRoutineEx routine registers an IoCompletion routine, which is called when the next-lower-level driver has completed the requested operation for the given IRP.
IoSetDeviceInterfacePropertyData	The IoSetDeviceInterfacePropertyData routine modifies the current value of a device interface property.

TITLE	DESCRIPTION
IoSetDeviceInterfaceState	The IoSetDeviceInterfaceState routine enables or disables an instance of a previously registered device interface class.
IoSetDevicePropertyData	The IoSetDevicePropertyData routine modifies the current setting for a device property.
IoSetIoPriorityHint	The IoSetIoPriorityHint routine sets the priority hint value for an IRP.
IoSetLinkShareAccess	The IoSetLinkShareAccess routine sets the access rights for link sharing the specified file object.
IoSetNextIrpStackLocation	The IoSetNextIrpStackLocation routine sets the IRP stack location in a driver-allocated IRP to that of the caller.
IoSetShareAccess	The IoSetShareAccess routine sets the access rights for sharing the given file object.
IoSetShareAccessEx	The IoSetShareAccessEx routine sets the access rights for sharing the specified file object.
IoSetStartIoAttributes	The IoSetStartIoAttributes routine sets attributes for the driver's StartIo routine.
IoSetTopLevelIrp	The IoSetTopLevelIrp routine sets the value of the TopLevelIrp field of the current thread.
IoSizeOfIrp	The IoSizeOfIrp routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.
IoSizeofWorkItem	The IoSizeofWorkItem routine returns the size, in bytes, of an IO_WORKITEM structure.
IoStartNextPacket	The IoStartNextPacket routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.
IoStartNextPacketByKey	The IoStartNextPacketByKey routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.
IoStartPacket	The IoStartPacket routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.
IoStartTimer	The IoStartTimer routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.
IoStopTimer	The IoStopTimer routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.

TITLE	DESCRIPTION
IoUninitializeWorkItem	The IoUninitializeWorkItem routine uninitializes a work item that was initialized by IoInitializeWorkItem.
IoUnregisterContainerNotification	The IoUnregisterContainerNotification routine cancels a container notification registration that was previously created by the IoRegisterContainerNotification routine.
IoUnregisterPlugPlayNotification	This routine is obsolete in Windows 7 and later versions of Windows. For more information, see the following Remarks section.The IoUnregisterPlugPlayNotification routine removes the registration of a driver's callback routine for a PnP event.
IoUnregisterPlugPlayNotificationEx	The IoUnregisterPlugPlayNotificationEx routine cancels the registration of a driver's callback routine for notifications of Plug and Play (PnP) events.
IoUnregisterShutdownNotification	The IoUnregisterShutdownNotification routine removes a registered driver from the shutdown notification queue.
IoUpdateLinkShareAccess	The IoUpdateLinkShareAccess routine updates the share access for the given file object, usually when the file is being opened.
IoUpdateShareAccess	The IoUpdateShareAccess routine updates the share access for the given file object, usually when the file is being opened.
IoValidateDeviceIoControlAccess	For more information, see the WdmlibIoValidateDeviceIoControlAccess function.
IoWithinStackLimits	The IoWithinStackLimits routine determines whether a region of memory is within the stack limit of the current thread.
IoWMIAllocateInstanceIds	The IoWMIAllocateInstanceIds routine allocates one or more instance IDs that are unique to the GUID.
IoWMIDeviceObjectToInstanceName	The IoWMIDeviceObjectToInstanceName routine determines the instance name for the WMI class instance implemented by the driver that is specified by a device object.
IoWMIDeviceObjectToProviderId	The IoWMIDeviceObjectToProviderId routine translates the specified device object into the corresponding WMI Provider ID.
IoWMIExecuteMethod	The IoWMIExecuteMethod routine runs a WMI class method on the specified WMI data block instance.
IoWMIHandleToInstanceName	The IoWMIHandleToInstanceName routine determines the instance name for the WMI class instance implemented by the driver that is specified by a file handle.
IoWMIOpenBlock	The IoWMIOpenBlock routine opens the WMI data block object for the specified WMI class.
IoWMIQueryAllData	The IoWMIQueryAllData routine returns all WMI data blocks that implement a given WMI class.

TITLE	DESCRIPTION
IoWMIQueryAllDataMultiple	The IoWMIQueryAllDataMultiple routine returns all WMI data blocks that implement one of a set of WMI classes.
IoWMIQuerySingleInstance	The IoWMIQuerySingleInstance routine returns the specified instance of a WMI data block.
IoWMIQuerySingleInstanceMultiple	The IoWMIQuerySingleInstanceMultiple routine returns all WMI data block instances that implement the specified WMI classes with the specified instance names.
IoWMIRegistrationControl	The IoWMIRegistrationControl routine registers or unregisters the caller as a WMI data provider for a specified device object.
IoWMISetNotificationCallback	The IoWMISetNotificationCallback routine registers a notification callback for a WMI event.
IoWMISetSingleInstance	The IoWMISetSingleInstance routine sets the values for properties within the data block instance that matches the specified WMI class and instance name.
IoWMISetSingleItem	The IoWMISetSingleItem routine sets a single property in the data block instance that matches the specified WMI class and instance name.
IoWMISuggestInstanceName	The IoWMISuggestInstanceName routine is used to request that WMI suggest a base name which a driver can use to build WMI instance names for the device.
IoWMIWriteEvent	The IoWMIWriteEvent routine delivers a given event to the user-mode WMI components for notification.
IoWriteErrorLogEntry	The IoWriteErrorLogEntry routine queues a given error log packet to the system error logging thread.
IsListEmpty	The IsListEmpty routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.
KdBreakPointWithStatus	The KdBreakPointWithStatus macro breaks into the kernel debugger and sends the value of Status to the debugger.
KdChangeOption	The KdChangeOption routine accesses and changes state in the kernel that is related to kernel debugging.
KdDisableDebugger	The KdDisableDebugger routine disables the kernel debugger.
KdEnableDebugger	The KdEnableDebugger routine re-enables the kernel debugger after a call to the KdDisableDebugger routine disables the kernel debugger.
KdPrint	The KdPrint macro sends a message to the kernel debugger.

TITLE	DESCRIPTION
KdPrintEx	The KdPrintEx macro sends a string to the kernel debugger if the conditions you specify are met.A call to KdPrintEx requires double parentheses.
KdRefreshDebuggerNotPresent	The KdRefreshDebuggerNotPresent macro refreshes the value of the KD_DEBUGGER_NOT_PRESENT global kernel variable.
KeAcquireSpinLock	The KeAcquireSpinLock routine acquires a spin lock so the caller can synchronize access to shared data in a multiprocessor-safe way by raising IRQL.
KeAcquireSpinLockAtDpcLevel	The KeAcquireSpinLockAtDpcLevel routine acquires a spin lock when the caller is already running at IRQL >= DISPATCH_LEVEL.
KeAddTriageDumpDataBlock	Adds a triage dump data block to a triage dump data block array.
KeAreAllApcsDisabled	The KeAreAllApcsDisabled routine indicates whether the calling thread is inside a guarded region or running at IRQL >= APC_LEVEL, which disables all APC delivery.
KeAreApcsDisabled	The KeAreApcsDisabled routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.
KeBugCheckEx	The KeBugCheckEx routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.
KeCancelTimer	The KeCancelTimer routine dequeues a timer object before the timer interval, if any was set, expires.
KeClearEvent	The KeClearEvent routine sets an event to a not-signaled state.
KeConvertAuxiliaryCounterToPerformanceCounter	The KeConvertAuxiliaryCounterToPerformanceCounter routine converts the specified auxiliary counter value into a performance counter value.
KeConvertPerformanceCounterToAuxiliaryCounter	The KeConvertPerformanceCounterToAuxiliaryCounter routine converts the specified performance counter value into an auxiliary counter value.
KeDelayExecutionThread	The KeDelayExecutionThread routine puts the current thread into an alertable or nonalertable wait state for a specified interval.
KeDeregisterBoundCallback	The KeDeregisterBoundCallback routine deregisters a user-mode bound exception callback registered by KeRegisterBoundCallback.
KeDeregisterBugCheckCallback	The KeDeregisterBugCheckCallback routine removes a callback routine that was registered by KeRegisterBugCheckCallback.

TITLE	DESCRIPTION
KeDeregisterBugCheckReasonCallback	The KeDeregisterBugCheckReasonCallback routine removes a callback routine that was registered by KeRegisterBugCheckReasonCallback.
KeDeregisterNmiCallback	The KeDeregisterNmiCallback routine deregisters a nonmaskable interrupt (NMI) callback registered by KeRegisterNmiCallback.
KeDeregisterProcessorChangeCallback	The KeDeregisterProcessorChangeCallback routine unregisters a callback function that was previously registered with the operating system by calling the KeRegisterProcessorChangeCallback routine.
KeEnterCriticalRegion	The KeEnterCriticalRegion routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.
KeEnterGuardedRegion	The KeEnterGuardedRegion routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.
KeFlushIoBuffers	The KeFlushIoBuffers routine flushes the memory region described by an MDL from caches of all processors.
KeFlushQueuedDpcs	The KeFlushQueuedDpcs routine returns after all queued DPCs on all processors have executed.
KefReleaseSpinLockFromDpcLevel	The KeReleaseSpinLockFromDpcLevel routine releases an executive spin lock without changing the IRQL.
KeGetCurrentIrql	The KeGetCurrentIrql routine returns the current IRQL.
KeGetCurrentNodeNumber	The KeGetCurrentNodeNumber routine gets the NUMA node number for the logical processor that the caller is running on.
KeGetCurrentProcessorNumberEx	The KeGetCurrentProcessorNumberEx routine gets the processor number of the logical processor that the caller is running on.
KeGetCurrentThread	The KeGetCurrentThread routine identifies the current thread.
KeGetProcessorIndexFromNumber	The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.
KeGetProcessorNumberFromIndex	The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.
KeGetRecommendedSharedDataAlignment	The KeGetRecommendedSharedDataAlignment routine returns the preferred alignment for memory structures that can be accessed by more than one processor.
KeInitializeCrashDumpHeader	The KeInitializeCrashDumpHeader routine supplies the header information the system requires for a crash dump file.

TITLE	DESCRIPTION
KeInitializeDeviceQueue	The KeInitializeDeviceQueue routine initializes a device queue object to a not-busy state.
KeInitializeDpc	The KeInitializeDpc routine initializes a DPC object, and registers a CustomDpc routine for that object.
KeInitializeEvent	The KeInitializeEvent routine initializes an event object as a synchronization (single waiter) or notification type event and sets it to a signaled or not-signaled state.
KeInitializeGuardedMutex	The KeInitializeGuardedMutex routine initializes a guarded mutex.
KeInitializeMutex	The KeInitializeMutex routine initializes a mutex object, setting it to a signaled state.
KeInitializeSemaphore	The KeInitializeSemaphore routine initializes a semaphore object with a specified count and specifies an upper limit that the count can attain.
KeInitializeSpinLock	The KeInitializeSpinLock routine initializes a variable of type KSPIN_LOCK.
KeInitializeThreadedDpc	The KeInitializeThreadedDpc routine initializes a threaded DPC object, and registers a CustomThreadedDpc routine for that object.
KeInitializeTimer	The KeInitializeTimer routine initializes a timer object.
KeInitializeTimerEx	The KeInitializeTimerEx routine initializes an extended kernel timer object.
KeInsertByKeyDeviceQueue	The KeInsertByKeyDeviceQueue routine acquires the spin lock for the specified DeviceQueue and queues an entry according to the specified sort-key value if the device queue is set to a busy state.
KeInsertDeviceQueue	The KeInsertDeviceQueue routine acquires the spin lock for the specified device queue object and, if the device queue is set to a busy state, queues the specified entry.
KeInsertQueueDpc	The KeInsertQueueDpc routine queues a DPC for execution.
KeIpiGenericCall	The KeIpiGenericCall routine causes the specified routine to run on all processors simultaneously.
KeIsExecutingDpc	Checks whether a DPC is being executed on current processor.
KeLeaveCriticalRegion	The KeLeaveCriticalRegion routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to KeEnterCriticalRegion.

TITLE	DESCRIPTION
KeLeaveGuardedRegion	The KeLeaveGuardedRegion routine exits a guarded region entered by KeEnterGuardedRegion.
KeLowerIrql	The KeLowerIrql routine restores the IRQL on the current processor to its original value.
KeMemoryBarrier	The KeMemoryBarrier routine creates a barrier at its position in the code—across which the compiler and the processor cannot move any operations.
KeQueryActiveGroupCount	The KeQueryActiveGroupCount routine returns the number of active processor groups in a multiprocessor system.
KeQueryActiveProcessorCount	The KeQueryActiveProcessorCount routine returns the number of currently active processors.
KeQueryActiveProcessorCountEx	The KeQueryActiveProcessorCountEx routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.
KeQueryActiveProcessors	The KeQueryActiveProcessors routine returns a bitmask of the currently active processors.
KeQueryAuxiliaryCounterFrequency	The KeQueryAuxiliaryCounterFrequency routine returns frequency of the auxiliary counter in units of Hz.
KeQueryDpcWatchdogInformation	The KeQueryDpcWatchdogInformation routine returns the deferred procedure call (DPC) watchdog timer values for the current processor.
KeQueryGroupAffinity	The KeQueryGroupAffinity routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.
KeQueryHighestNodeNumber	The KeQueryHighestNodeNumber routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryInterruptTime	The KeQueryInterruptTime routine returns the current value of the system interrupt time count, with accuracy to within system clock tick.
KeQueryInterruptTimePrecise	The KeQueryInterruptTimePrecise routine returns the current value of the system interrupt time count, with accuracy to within a microsecond.
KeQueryLogicalProcessorRelationship	The KeQueryLogicalProcessorRelationship routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.
KeQueryMaximumGroupCount	The KeQueryMaximumGroupCount routine returns the maximum number of groups in a multiprocessor system.
KeQueryMaximumProcessorCount	The KeQueryMaximumProcessorCount routine returns the maximum number of processors.

TITLE	DESCRIPTION
KeQueryMaximumProcessorCountEx	The KeQueryMaximumProcessorCountEx routine returns the maximum number of logical processors in a specified group in a multiprocessor system.
KeQueryNodeActiveAffinity	The KeQueryNodeActiveAffinity routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.
KeQueryNodeMaximumProcessorCount	The KeQueryNodeMaximumProcessorCount routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.
KeQueryPerformanceCounter	The KeQueryPerformanceCounter routine retrieves the current value and frequency of the performance counter.Use KeQueryPerformanceCounter to acquire high resolution (<1us) time stamps for time interval measurements.
KeQueryPriorityThread	The KeQueryPriorityThread routine returns the current priority of a particular thread.
KeQueryRuntimeThread	The KeQueryRuntimeThread routine reports the accumulated kernel-mode and user-mode run time of a thread, in clock ticks.
KeQuerySystemTime	The KeQuerySystemTime routine obtains the current system time.
KeQuerySystemTimePrecise	The KeQuerySystemTimePrecise routine retrieves the current system time, and is more precise than the KeQuerySystemTime routine.
KeQueryTickCount	The KeQueryTickCount routine maintains a count of the interval timer interrupts that have occurred since the system was booted.
KeQueryTimeIncrement	The KeQueryTimeIncrement routine returns the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.
KeQueryTotalCycleTimeThread	The KeQueryTotalCycleTimeThread routine returns the accumulated cycle time for the specified thread.
KeQueryUnbiasedInterruptTime	The KeQueryUnbiasedInterruptTime routine returns the current value of the system interrupt time count.
KeRaiseIrql	The KeRaiseIrql routine raises the hardware priority to the specified IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

TITLE	DESCRIPTION
KeRaiseIrqlToDpcLevel	The KeRaiseIrqlToDpcLevel routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
KeReadStateEvent	The KeReadStateEvent routine returns the current state, signaled or not-signaled, of an event object.
KeReadStateMutex	The KeReadStateMutex routine returns the current state, signaled or not-signaled, of the specified mutex object.
KeReadStateSemaphore	The KeReadStateSemaphore routine returns the current state, signaled or not-signaled, of the specified semaphore object.
KeReadStateTimer	The KeReadStateTimer routine reads the current state of a timer object.
KeRegisterBoundCallback	The KeRegisterBoundCallback routine registers a routine to be called whenever a user-mode bound exception occurs.
KeRegisterBugCheckCallback	The KeRegisterBugCheckCallback routine registers a BugCheckCallback routine, which executes when the operating system issues a bug check.
KeRegisterBugCheckReasonCallback	The KeRegisterBugCheckReasonCallback routine registers a BugCheckDumpIoCallback, BugCheckSecondaryDumpDataCallback, or BugCheckAddPagesCallback routine, which executes when the operating system issues a bug check.
KeRegisterNmiCallback	The KeRegisterNmiCallback routine registers a routine to be called whenever a nonmaskable interrupt (NMI) occurs.
KeRegisterProcessorChangeCallback	The KeRegisterProcessorChangeCallback routine registers a callback function with the operating system so that the operating system will notify the driver when a new processor is added to the hardware partition.
KeReleaseGuardedMutex	The KeReleaseGuardedMutex routine releases a guarded mutex acquired with KeAcquireGuardedMutex or KeTryToAcquireGuardedMutex.
KeReleaseGuardedMutexUnsafe	The KeReleaseGuardedMutexUnsafe routine releases a guarded mutex acquired by KeAcquireGuardedMutexUnsafe.
KeReleaseInStackQueuedSpinLock	The KeReleaseInStackQueuedSpinLock routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLock.
KeReleaseInStackQueuedSpinLockForDpc	The KeReleaseInStackQueuedSpinLockForDpc routine releases a queued spin lock that was acquired by calling KeAcquireInStackQueuedSpinLockForDpc.
KeReleaseInStackQueuedSpinLockFromDpcLevel	The KeReleaseInStackQueuedSpinLockFromDpcLevel routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLockAtDpcLevel.

TITLE	DESCRIPTION
KeReleaseInterruptSpinLock	The KeReleaseInterruptSpinLock routine releases an interrupt spin lock acquired by KeAcquireInterruptSpinLock.
KeReleaseMutex	The KeReleaseMutex routine releases a mutex object, and specifies whether the caller is to call one of the KeWaitXxx routines as soon as KeReleaseMutex returns control.
KeReleaseSemaphore	The KeReleaseSemaphore routine releases the specified semaphore object.
KeReleaseSpinLock	The KeReleaseSpinLock routine releases a spin lock and restores the original IRQL at which the caller was running.
KeReleaseSpinLockForDpc	The KeReleaseSpinLockForDpc routine releases a spin lock that was acquired by calling KeAcquireSpinLockForDpc.
KeReleaseSpinLockFromDpcLevel	The KeReleaseSpinLockFromDpcLevel routine releases an executive spin lock without changing the IRQL.
KeRemoveByKeyDeviceQueue	The KeRemoveByKeyDeviceQueue routine removes an entry, selected according to a sort key value, from the specified device queue.
KeRemoveDeviceQueue	The KeRemoveDeviceQueue routine removes an entry from the head of a specified device queue.
KeRemoveEntryDeviceQueue	The KeRemoveEntryDeviceQueue routine returns whether the specified entry is in the device queue and removes it, if it was queued, from the device queue.
KeRemoveQueueDpc	The KeRemoveQueueDpc routine removes the specified DPC object from the system DPC queue.
KeResetEvent	The KeResetEvent routine resets a specified event object to a not-signaled state and returns the previous state of that event object.
KeRestoreExtendedProcessorState	The KeRestoreExtendedProcessorState routine restores extended processor state information that was previously saved.
KeRestoreFloatingPointState	The KeRestoreFloatingPointState routine restores the nonvolatile floating-point context saved by the preceding call to KeSaveFloatingPointState.
KeRevertToUserAffinityThreadEx	The KeRevertToUserAffinityThreadEx routine restores the previous affinity of the current thread.
KeRevertToUserGroupAffinityThread	The KeRevertToUserGroupAffinityThread routine restores the group affinity of the calling thread to its original value at the time that the thread was created.
KeSaveExtendedProcessorState	The KeSaveExtendedProcessorState routine saves extended processor state information.

TITLE	DESCRIPTION
KeSaveFloatingPointState	The KeSaveFloatingPointState routine saves the nonvolatile floating-point context so the caller can carry out floating-point operations.
KeSetCoalescableTimer	The KeSetCoalescableTimer routine sets the initial expiration time and period of a timer object and specifies how much delay can be tolerated in the expiration times.
KeSetEvent	The KeSetEvent routine sets an event object to a signaled state if the event was not already signaled, and returns the previous state of the event object.
KeSetImportanceDpc	The KeSetImportanceDpc routine specifies how soon the DPC routine is run.
KeSetPriorityThread	The KeSetPriorityThread routine sets the run-time priority of a driver-created thread.
KeSetSystemAffinityThread	The KeSetSystemAffinityThread routine sets the system affinity of the current thread.
KeSetSystemAffinityThreadEx	The KeSetSystemAffinityThreadEx routine sets the system affinity of the current thread.
KeSetSystemGroupAffinityThread	The KeSetSystemGroupAffinityThread routine changes the group number and affinity mask of the calling thread.
KeSetTargetProcessorDpc	The KeSetTargetProcessorDpc routine specifies the processor that a DPC routine will be run on.
KeSetTargetProcessorDpcEx	The KeSetTargetProcessorDpcEx routine specifies the processor that a DPC routine will run on.
KeSetTimer	The KeSetTimer routine sets the absolute or relative interval at which a timer object is to be set to a signaled state and, optionally, supplies a CustomTimerDpc routine to be executed when that interval expires.
KeSetTimerEx	The KeSetTimerEx routine sets the absolute or relative interval at which a timer object is to be set to a signaled state, optionally supplies a CustomTimerDpc routine to be executed when that interval expires, and optionally supplies a recurring interval for the timer.
KeStallExecutionProcessor	The KeStallExecutionProcessor routine stalls the caller on the current processor for a specified time interval.
KeSynchronizeExecution	The KeSynchronizeExecution routine synchronizes the execution of the specified routine with the interrupt service routine (ISR) that is assigned to a set of one or more interrupt objects.
KeTestSpinLock	The KeTestSpinLock routine tests for the availability of a spin lock.

TITLE	DESCRIPTION
KeTryToAcquireSpinLockAtDpcLevel	The KeTryToAcquireSpinLockAtDpcLevel routine attempts to acquire a spin lock at DISPATCH_LEVEL.
KeWaitForMultipleObjects	The KeWaitForMultipleObjects routine puts the current thread into an alertable or nonalertable wait state until any or all of a number of dispatcher objects are set to a signaled state or (optionally) until the wait times out.
KeWaitForSingleObject	The KeWaitForSingleObject routine puts the current thread into a wait state until the given dispatcher object is set to a signaled state or (optionally) until the wait times out.
KzLowerIrql	Restores the IRQL on the current processor to its original value.
KzRaiseIrql	Raises the hardware priority to the specified IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
MmAdvanceMdl	The MmAdvanceMdl routine advances the beginning of an MDL's virtual memory range by the specified number of bytes.
MmAllocateContiguousMemory	The MmAllocateContiguousMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCache	The MmAllocateContiguousMemorySpecifyCache routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousMemorySpecifyCacheNode	The MmAllocateContiguousMemorySpecifyCacheNode routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateContiguousNodeMemory	The MmAllocateContiguousNodeMemory routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.
MmAllocateMappingAddress	The MmAllocateMappingAddress routine reserves a range of system virtual address space of the specified size.
MmAllocateMdlForIoSpace	The MmAllocateMdlForIoSpace routine allocates an MDL and initializes this MDL to describe a set of physical address ranges in I/O address space.
MmAllocateNodePagesForMdlEx	The MmAllocateNodePagesForMdlEx routine allocates nonpaged physical memory from an ideal node, and allocates an MDL structure to describe this memory.
MmAllocatePagesForMdl	The MmAllocatePagesForMdl routine allocates zero-filled, nonpaged, physical memory pages to an MDL.
MmAllocatePagesForMdlEx	The MmAllocatePagesForMdlEx routine allocates nonpaged, physical memory pages to an MDL.

TITLE	DESCRIPTION
MmBuildMdlForNonPagedPool	The MmBuildMdlForNonPagedPool routine receives an MDL that specifies a nonpaged virtual memory buffer, and updates it to describe the underlying physical pages.
MmFreeContiguousMemory	The MmFreeContiguousMemory routine releases a range of physically contiguous memory that was allocated by an MmAllocateContiguousMemoryXxx routine.
MmFreeContiguousMemorySpecifyCache	The MmFreeContiguousMemorySpecifyCache routine frees a buffer that was allocated by an MmAllocateContiguousMemorySpecifyCacheXxx routine.
MmFreeMappingAddress	The MmFreeMappingAddress routine frees a range of virtual memory reserved by the MmAllocateMappingAddress routine.
MmFreePagesFromMdl	The MmFreePagesFromMdl routine frees all the physical pages that are described by an MDL that was created by the MmAllocatePagesForMdl routine.
MmGetMdlByteCount	The MmGetMdlByteCount macro returns the length, in bytes, of the buffer described by the specified MDL.
MmGetSystemAddressForMdl	The MmGetSystemAddressForMdl routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me.
MmGetSystemRoutineAddress	The MmGetSystemRoutineAddress routine returns a pointer to a function specified by SystemRoutineName.
MmIsDriverSuspectForVerifier	The MmIsDriverSuspectForVerifier routine indicates whether the driver represented by the specified driver object is in the list of drivers that are selected to be verified by Driver Verifier.
MmIsDriverVerifying	The MmIsDriverVerifying routine indicates whether the kernel-mode driver that is identified by the specified driver object is being verified or calls a driver that is being verified by Driver Verifier.
MmIsDriverVerifyingByAddress	The MmIsDriverVerifyingByAddress routine checks whether the kernel-mode driver that is identified by the specified image address is being verified or calls a driver that is being verified by Driver Verifier.
MmLockPagableCodeSection	The MmLockPagableCodeSection routine locks a section of driver code, containing a set of driver routines marked with a special compiler directive, into system space.
MmLockPagableDataSection	The MmLockPagableDataSection routine locks an entire section of driver data into system space.
MmMapIoSpace	The MmMapIoSpace routine maps the given physical address range to nonpaged system space.

TITLE	DESCRIPTION
MmMapIoSpaceEx	The MmMapIoSpaceEx routine maps the given physical address range to non-paged system space using the specified page protection.
MmMapLockedPages	The MmMapLockedPages routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me.
MmMapLockedPagesSpecifyCache	The MmMapLockedPagesSpecifyCache routine maps the physical pages that are described by an MDL to a virtual address, and enables the caller to specify the cache attribute that is used to create the mapping.
MmMapLockedPagesWithReservedMapping	The MmMapLockedPagesWithReservedMapping routine maps all or part of an address range that was previously reserved by the MmAllocateMappingAddress routine.
MmMapMdl	This function maps physical pages described by a memory descriptor list (MDL) into the system virtual address space.
MmPageEntireDriver	The MmPageEntireDriver routine causes all of a driver's code and data to be made pageable, overriding the attributes of the various sections that make up the driver's image.
MmProbeAndLockPages	The MmProbeAndLockPages routine probes the specified virtual memory pages, makes them resident, and locks them in memory.
MmProbeAndLockSelectedPages	The MmProbeAndLockSelectedPages routine probes the selected virtual memory pages, makes them resident, and locks them in memory.
MmProtectMdlSystemAddress	The MmProtectMdlSystemAddress routine sets the protection type for a memory address range.
MmQuerySystemSize	The MmQuerySystemSize routine returns an estimate of the amount of memory in the system.
MmResetDriverPaging	The MmResetDriverPaging routine resets the pageable status of a driver's sections to that specified when the driver was compiled.
MmSizeOfMdl	The MmSizeOfMdl routine returns the number of bytes to allocate for an MDL describing a given address range.
MmUnlockPagableImageSection	The MmUnlockPagableImageSection routine releases a section of driver code or driver data, previously locked into system space with MmLockPagableCodeSection, MmLockPagableDataSection or MmLockPagableSectionByHandle, so the section can be paged out again.
MmUnlockPages	The MmUnlockPages routine unlocks the physical pages that are described by the specified memory descriptor list (MDL).

TITLE	DESCRIPTION
MmUnmapIoSpace	The MmUnmapIoSpace routine unmaps a specified range of physical addresses previously mapped by MmMapIoSpace.
MmUnmapLockedPages	The MmUnmapLockedPages routine releases a mapping that was set up by a preceding call to the MmMapLockedPages or MmMapLockedPagesSpecifyCache routine.
MmUnmapReservedMapping	The MmUnmapReservedMapping routine unmaps a memory buffer that was mapped by the MmMapLockedPagesWithReservedMapping routine.
NtCommitComplete	The ZwCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction's data.
NtCommitEnlistment	The ZwCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
NtCommitTransaction	The ZwCommitTransaction routine initiates a commit operation for a specified transaction.
NtCreateEnlistment	The ZwCreateEnlistment routine creates a new enlistment object for a transaction.
NtCreateResourceManager	The ZwCreateResourceManager routine creates a resource manager object.
NtCreateTransaction	The ZwCreateTransaction routine creates a transaction object.
NtCreateTransactionManager	The ZwCreateTransactionManager routine creates a new transaction manager object.
NtEnumerateTransactionObject	The ZwEnumerateTransactionObject routine enumerates the KTM objects on a computer.
NtGetNotificationResourceManager	The ZwGetNotificationResourceManager routine retrieves the next transaction notification from a specified resource manager's notification queue.
NtOpenEnlistment	The ZwOpenEnlistment routine obtains a handle to an existing enlistment object.
NtOpenResourceManager	The ZwOpenResourceManager routine returns a handle to an existing resource manager object.
NtOpenTransaction	The ZwOpenTransaction routine obtains a handle to an existing transaction object.
NtOpenTransactionManager	The ZwOpenTransactionManager routine obtains a handle to an existing transaction manager object.
NtPowerInformation	The ZwPowerInformation routine sets or retrieves system power information.

TITLE	DESCRIPTION
NtPrepareComplete	The ZwPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
NtPrepareEnlistment	The ZwPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
NtPrePrepareComplete	The ZwPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.
NtPrePrepareEnlistment	The ZwPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
NtQueryInformationEnlistment	The ZwQueryInformationEnlistment routine retrieves information about a specified enlistment object.
NtQueryInformationResourceManager	The ZwQueryInformationResourceManager routine retrieves information about a specified resource manager object.
NtQueryInformationTransaction	The ZwQueryInformationTransaction routine retrieves information about a specified transaction.
NtQueryInformationTransactionManager	The ZwQueryInformationTransactionManager routine retrieves information about a specified transaction manager object.
NtReadOnlyEnlistment	The ZwReadOnlyEnlistment routine sets a specified enlistment to be read-only.
NtRecoverEnlistment	The ZwRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.
NtRecoverResourceManager	The ZwRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
NtRecoverTransactionManager	The ZwRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
NtRenameTransactionManager	The NtRenameTransactionManager routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.
NtRollbackComplete	The ZwRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
NtRollbackEnlistment	The ZwRollbackEnlistment routine rolls back the transaction that is associated with a specified enlistment.
NtRollbackTransaction	The ZwRollbackTransaction routine initiates a rollback operation for a specified transaction.

TITLE	DESCRIPTION
NtRollforwardTransactionManager	The ZwRollforwardTransactionManager routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.
NtSetInformationEnlistment	The ZwSetInformationEnlistment routine sets information for a specified enlistment object.
NtSetInformationResourceManager	The ZwSetInformationResourceManager routine is not used.
NtSetInformationTransaction	The ZwSetInformationTransaction routine sets information for a specified transaction.
NtSetInformationTransactionManager	Do not call this routine from kernel-mode code.
NtSinglePhaseReject	The ZwSinglePhaseReject routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.
ObCloseHandle	The ObCloseHandle routine closes an object handle.
ObDereferenceObject	The ObDereferenceObject routine decrements the given object's reference count and performs retention checks.
ObDereferenceObjectDeferDelete	The ObDereferenceObjectDeferDelete routine decrements the reference count for the given object, checks for object retention, and avoids deadlocks.
ObDereferenceObjectDeferDeleteWithTag	The ObDereferenceObjectDeferDeleteWithTag routine decrements the reference count for the specified object, defers deletion of the object to avoid deadlocks, and writes a four-byte tag value to the object to support object reference tracing.
ObDereferenceObjectWithTag	The ObDereferenceObjectWithTag routine decrements the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.
ObfReferenceObject	The ObReferenceObject routine increments the reference count to the given object.
ObGetObjectSecurity	The ObGetObjectSecurity routine gets the security descriptor for a given object.
ObReferenceObject	The ObReferenceObject routine increments the reference count to the given object.
ObReferenceObjectByHandle	The ObReferenceObjectByHandle routine provides access validation on the object handle, and, if access can be granted, returns the corresponding pointer to the object's body.
ObReferenceObjectByHandleWithTag	The ObReferenceObjectByHandleWithTag routine increments the reference count of the object that is identified by the specified handle, and writes a four-byte tag value to the object to support object reference tracing.

TITLE	DESCRIPTION
ObReferenceObjectByPointer	The ObReferenceObjectByPointer routine increments the pointer reference count for a given object.
ObReferenceObjectByPointerWithTag	The ObReferenceObjectByPointerWithTag routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.
ObReferenceObjectSafe	The ObReferenceObjectSafe function increments the reference count for an object and determines if it is safe to use the object. It returns FALSE if the object is being deleted or TRUE if it's safe to use the object further.
ObReferenceObjectWithTag	The ObReferenceObjectWithTag routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.
ObRegisterCallbacks	The ObRegisterCallbacks routine registers a list of callback routines for thread, process, and desktop handle operations.
ObReleaseObjectSecurity	The ObReleaseObjectSecurity routine is the reciprocal to ObGetObjectSecurity.
ObUnRegisterCallbacks	The ObUnRegisterCallbacks routine unregisters a set of callback routines that were registered with the ObRegisterCallbacks routine.
PcwAddInstance	The PcwAddInstance function adds the specified instance of the counter set to the consumer buffer.
PcwCloseInstance	The PcwCloseInstance function closes the specified instance of the counter set.
PcwCreateInstance	The PcwCreateInstance function creates a new instance for the specified registered counter set.
PcwRegister	The PcwRegister function registers the caller as a provider of the specified counter set.
PcwUnregister	The PcwUnregister function unregisters the provider of the specified counter set.
PoCallDriver	The PoCallDriver routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoClearPowerRequest	The PoClearPowerRequest routine decrements the count for the specified power request type.
PoCreatePowerRequest	The PoCreatePowerRequest routine creates a power request object.
PoDeletePowerRequest	The PoDeletePowerRequest routine deletes a power request object.

TITLE	DESCRIPTION
PoEndDeviceBusy	The PoEndDeviceBusy routine marks the end of a period of time in which the device is busy.
PoFxActivateComponent	The PoFxActivateComponent routine increments the activation reference count on the specified component.
PoFxCompleteDevicePowerNotRequired	The PoFxCompleteDevicePowerNotRequired routine notifies the power management framework (PoFx) that the calling driver has completed its response to a call to the driver's DevicePowerNotRequiredCallback callback routine.
PoFxCompleteIdleCondition	The PoFxCompleteIdleCondition routine informs the power management framework (PoFx) that the specified component has completed a pending change to the idle condition.
PoFxCompleteIdleState	The PoFxCompleteIdleState routine informs the power management framework (PoFx) that the specified component has completed a pending change to an Fx state.
PoFxIdleComponent	The PoFxIdleComponent routine decrements the activation reference count on the specified component.
PoFxIssueComponentPerfStateChange	The PoFxIssueComponentPerfStateChange routine submits a request to place a device component in a particular performance state.
PoFxIssueComponentPerfStateChangeMultiple	The PoFxIssueComponentPerfStateChangeMultiple routine submits a request to change the performance states in multiple performance state sets simultaneously for a device component.
PoFxNotifySurprisePowerOn	The PoFxNotifySurprisePowerOn routine notifies the power management framework (PoFx) that a device was turned on as a side effect of supplying power to some other device.
PoFxPowerControl	The PoFxPowerControl routine sends a power control request to the power management framework (PoFx).
PoFxPowerOnCrashdumpDevice	The PoFxPowerOnCrashdumpDevice routine requests that a crash-dump device be turned on.
PoFxQueryCurrentComponentPerfState	The PoFxQueryCurrentComponentPerfState routine retrieves the active performance state in a component's performance state set.
PoFxRegisterComponentPerfStates	The PoFxRegisterComponentPerfStates routine registers a device component for performance state management by the power management framework (PoFx).
PoFxRegisterCrashdumpDevice	The PoFxRegisterCrashdumpDevice routine registers a crash-dump device.
PoFxRegisterDevice	The PoFxRegisterDevice routine registers a device with the power management framework (PoFx).

TITLE	DESCRIPTION
PoFxReportDevicePoweredOn	The PoFxReportDevicePoweredOn routine notifies the power management framework (PoFx) that the device completed the requested transition to the D0 (fully on) power state.
PoFxSetComponentLatency	The PoFxSetComponentLatency routine specifies the maximum latency that can be tolerated in the transition from the idle condition to the active condition in the specified component.
PoFxSetComponentResidency	The PoFxSetComponentResidency routine sets the estimated time for how long a component is likely to remain idle after the component enters the idle condition.
PoFxSetComponentWake	The PoFxSetComponentWake routine indicates whether the driver arms the specified component to wake whenever the component enters the idle condition.
PoFxSetDeviceIdleTimeout	The PoFxSetDeviceIdleTimeout routine specifies the minimum time interval from when the last component of the device enters the idle condition to when the power management framework (PoFx) calls the driver's DevicePowerNotRequiredCallback routine.
PoFxSetTargetDripsDevicePowerState	This routine is called to notify the power manager of the device's target device power state for DRIPS. The driver can override the DRIPS constraint provided by the PEP.
PoFxStartDevicePowerManagement	The PoFxStartDevicePowerManagement routine completes the registration of a device with the power management framework (PoFx) and starts device power management.
PoFxUnregisterDevice	The PoFxUnregisterDevice routine removes the registration of a device from the power management framework (PoFx).
PoGetSystemWake	The PoGetSystemWake routine determines whether a specified IRP has been marked as waking the system from a sleeping state.
PopEntryList	The PopEntryList routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.
PoQueryWatchdogTime	The PoQueryWatchdogTime routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.
PoRegisterDeviceForIdleDetection	The PoRegisterDeviceForIdleDetection routine enables or cancels idle detection and sets idle time-out values for a device.
PoRegisterPowerSettingCallback	The PoRegisterPowerSettingCallback routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.
PoRegisterSystemState	The PoRegisterSystemState routine registers the system as busy due to certain activity.

TITLE	DESCRIPTION
PoRequestPowerIrp	The PoRequestPowerIrp routine allocates a power IRP and sends it to the top driver in the device stack for the specified device.
PoSetDeviceBusyEx	The PoSetDeviceBusyEx routine notifies the power manager that the device associated with the specified idle counter is busy.
PoSetPowerRequest	The PoSetPowerRequest routine increments the count for the specified power request type.
PoSetPowerState	The PoSetPowerState routine notifies the system of a change in the device power state for a device.
PoSetSystemState	Drivers call the PoSetSystemState routine to indicate that the system is active.
PoSetSystemWake	The PoSetSystemWake routine marks the specified IRP as one that contributed to waking the system from a sleep state.
PoStartDeviceBusy	The PoStartDeviceBusy routine marks the start of a period of time in which the device is busy.
PoStartNextPowerIrp	The PoStartNextPowerIrp routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.).
PoUnregisterPowerSettingCallback	The PoUnregisterPowerSettingCallback routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.
PoUnregisterSystemState	The PoUnregisterSystemState routine cancels a system state registration created by PoRegisterSystemState.
ProbeForRead	The ProbeForRead routine checks that a user-mode buffer actually resides in the user portion of the address space, and is correctly aligned.
ProbeForWrite	The ProbeForWrite routine checks that a user-mode buffer actually resides in the user-mode portion of the address space, is writable, and is correctly aligned.
PsCreateSystemThread	The PsCreateSystemThread routine creates a system thread that executes in kernel mode and returns a handle for the thread.
PsGetCurrentThread	The PsGetCurrentThread routine identifies the current thread.
PsGetVersion	This function is obsolete in Windows XP and later versions of the Windows operating system. Use RtlGetVersion instead.PsGetVersion returns caller-selected information about the current version of the NT-based operating system.

TITLE	DESCRIPTION
PsQueryTotalCycleTimeProcess	The PsQueryTotalCycleTimeProcess routine returns the accumulated cycle time for the specified process.
PsTerminateSystemThread	The PsTerminateSystemThread routine terminates the current system thread.
PushEntryList	The PushEntryList routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
READ_PORT_BUFFER_UCHAR	The READ_PORT_BUFFER_UCHAR routine reads a number of bytes from the specified port address into a buffer.
READ_PORT_BUFFER_ULONG	The READ_PORT_BUFFER_ULONG routine reads a number of ULONG values from the specified port address into a buffer.
READ_PORT_BUFFER_USHORT	The READ_PORT_BUFFER_USHORT routine reads a number of USHORT values from the specified port address into a buffer.
READ_PORT_UCHAR	The READ_PORT_UCHAR routine reads a byte from the specified port address.
READ_PORT_ULONG	The READ_PORT_ULONG routine reads a ULONG value from the specified port address.
READ_PORT_USHORT	The READ_PORT_USHORT routine reads a USHORT value from the specified port address.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG64	The READ_REGISTER_BUFFER_ULONG64 function reads a number of ULONG64 values from the specified register address into a buffer.
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG64	The READ_REGISTER_ULONG64 function reads a ULONG64 value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.

TITLE	DESCRIPTION
RemoveEntryList	The RemoveEntryList routine removes an entry from a doubly linked list of LIST_ENTRY structures.
RemoveHeadList	The RemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
RemoveTailList	The RemoveTailList routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.
RtlAnsiStringToUnicodeSize	The RtlAnsiStringToUnicodeSize routine returns the number of bytes required to hold an ANSI string converted into a Unicode string.
RtlAnsiStringToUnicodeString	RtlAnsiStringToUnicodeString converts the given ANSI source string into a Unicode string.
RtlAppendUnicodeStringToString	The RtlAppendUnicodeStringToString routine concatenates two Unicode strings.
RtlAppendUnicodeToString	The RtlAppendUnicodeToString routine concatenates the supplied Unicode string to a buffered Unicode string.
RtlAreBitsClear	The RtlAreBitsClear routine determines whether a given range of bits within a bitmap variable is clear.
RtlAreBitsSet	The RtlAreBitsSet routine determines whether a given range of bits within a bitmap variable is set.
RtlCheckBit	The RtlCheckBit routine determines whether a particular bit in a given bitmap variable is clear or set.
RtlCheckRegistryKey	The RtlCheckRegistryKey routine checks for the existence of a given named key in the registry.
RtlClearAllBits	The RtlClearAllBits routine sets all bits in a given bitmap variable to zero.
RtlClearBit	The RtlClearBit routine sets the specified bit in a bitmap to zero.
RtlClearBits	The RtlClearBits routine sets all bits in the specified range of bits in the bitmap to zero.
RtlCmDecodeMemIoResource	The RtlCmDecodeMemIoResource routine provides the starting address and length of a CM_PARTIAL_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.
RtlCmEncodeMemIoResource	The RtlCmEncodeMemIoResource routine updates a CM_PARTIAL_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.
RtlCompareMemory	The RtlCompareMemory routine compares two blocks of memory and returns the number of bytes that match.

TITLE	DESCRIPTION
RtlCompareUnicodeString	The RtlCompareUnicodeString routine compares two Unicode strings.
RtlConvertLongToLargeInteger	The RtlConvertLongToLargeInteger routine converts the input signed integer to a signed large integer.
RtlConvertUlongToLargeInteger	The RtlConvertUlongToLargeInteger routine converts the input unsigned integer to a signed large integer. For Windows XP and later versions of Windows, do not use this routine; use the native support for __int64.
RtlCopyMemory	The RtlCopyMemory routine copies the contents of a source memory block to a destination memory block.
RtlCopyMemoryNonTemporal	This function copies from one buffer to another using non-temporal moves that do not pollute the cache. The buffers shouldn't overlap.
RtlCopyUnicodeString	The RtlCopyUnicodeString routine copies a source string to a destination string.
RtlCreateRegistryKey	The RtlCreateRegistryKey routine adds a key object in the registry along a given relative path.
RtlCreateSecurityDescriptor	The RtlCreateSecurityDescriptor routine initializes a new absolute-format security descriptor. On return, the security descriptor is initialized with no system ACL, no discretionary ACL, no owner, no primary group, and all control flags set to zero.
RtlDeleteRegistryValue	The RtlDeleteRegistryValue routine removes the specified entry name and the associated values from the registry along the given relative path.
RtlDowncaseUnicodeChar	The RtlDowncaseUnicodeChar routine converts the specified Unicode character to lowercase.
RtlEqualMemory	The RtlEqualMemory routine compares two blocks of memory to determine whether the specified number of bytes are identical.
RtlEqualUnicodeString	The RtlEqualUnicodeString routine compares two Unicode strings to determine whether they are equal.
RtlFillMemory	The RtlFillMemory routine fills a block of memory with the specified fill value.
RtlFindClearBits	The RtlFindClearBits routine searches for a range of clear bits of a requested size within a bitmap.
RtlFindClearBitsAndSet	The RtlFindClearBitsAndSet routine searches for a range of clear bits of a requested size within a bitmap and sets all bits in the range when it has been located.

TITLE	DESCRIPTION
RtlFindClearRuns	The RtlFindClearRuns routine finds the specified number of runs of clear bits within a given bitmap.
RtlFindFirstRunClear	The RtlFindFirstRunClear routine searches for the initial contiguous range of clear bits within a given bitmap.
RtlFindLastBackwardRunClear	The RtlFindLastBackwardRunClear routine searches a given bitmap for the preceding clear run of bits, starting from the specified index position.
RtlFindLeastSignificantBit	The RtlFindLeastSignificantBit routine returns the zero-based position of the least significant nonzero bit in its parameter.
RtlFindLongestRunClear	The RtlFindLongestRunClear routine searches for the largest contiguous range of clear bits within a given bitmap.
RtlFindMostSignificantBit	The RtlFindMostSignificantBit routine returns the zero-based position of the most significant nonzero bit in its parameter.
RtlFindNextForwardRunClear	The RtlFindNextForwardRunClear routine searches a given bitmap variable for the next clear run of bits, starting from the specified index position.
RtlFindSetBits	The RtlFindSetBits routine searches for a range of set bits of a requested size within a bitmap.
RtlFindSetBitsAndClear	The RtlFindSetBitsAndClear routine searches for a range of set bits of a requested size within a bitmap and clears all bits in the range when it has been located.
RtlFreeAnsiString	The RtlFreeAnsiString routine releases storage that was allocated by RtlUnicodeStringToAnsiString.
RtlFreeUnicodeString	The RtlFreeUnicodeString routine releases storage that was allocated by RtlAnsiStringToUnicodeString or RtlUpcaseUnicodeString.
RtlGetVersion	The RtlGetVersion routine returns version information about the currently running operating system.
RtlGUIDFromString	The RtlGUIDFromString routine converts the given Unicode string to a GUID in binary format.
RtlHashUnicodeString	The RtlHashUnicodeString routine creates a hash value from a given Unicode string and hash algorithm.
RtlInitAnsiString	The RtlInitAnsiString routine initializes a counted string of ANSI characters.
RtlInitializeBitMap	The RtlInitializeBitMap routine initializes the header of a bitmap variable.
RtlInitString	The RtlInitString routine initializes a counted string of 8-bit characters.

TITLE	DESCRIPTION
RtlInitStringEx	The RtlInitStringEx routine initializes a counted string of 8-bit characters.
RtlInitUnicodeString	For more information, see the WdmlibRtlInitUnicodeStringEx function.
RtlInt64ToUnicodeString	The RtlInt64ToUnicodeString routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.
RtlIntegerToUnicodeString	The RtlIntegerToUnicodeString routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.
RtlIntPtrToUnicodeString	The RtlIntPtrToUnicodeString routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.
RtlIoDecodeMemIoResource	The RtlIoDecodeMemIoResource routine provides the address information that is contained in an IO_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.
RtlIoEncodeMemIoResource	The RtlIoEncodeMemIoResource routine updates an IO_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.
RtlIsNtDdiVersionAvailable	The RtlIsNtDdiVersionAvailable routine determines if a specified version of the Microsoft Windows device driver interface (DDI) is available.
RtlIsServicePackVersionInstalled	The RtlIsServicePackVersionInstalled routine determines if a specified service pack version of the Microsoft Windows device driver interface (DDI) is installed.
RtlLengthSecurityDescriptor	The RtlLengthSecurityDescriptor routine returns the size of a given security descriptor.
RtlMoveMemory	The RtlMoveMemory routine copies the contents of a source memory block to a destination memory block, and supports overlapping source and destination memory blocks.
RtlNumberOfClearBits	The RtlNumberOfClearBits routine returns a count of the clear bits in a given bitmap variable.
RtlNumberOfSetBits	The RtlNumberOfSetBits routine returns a count of the set bits in a given bitmap variable.
RtlNumberOfSetBitsUlongPtr	The RtlNumberOfSetBitsUlongPtr routine returns the number of bits in the specified ULONG_PTR integer value that are set to one.
RtlPrefetchMemoryNonTemporal	The RtlPrefetchMemoryNonTemporal routine provides a hint to the processor that a buffer should be temporarily moved into the processor cache.

TITLE	DESCRIPTION
RtlQueryRegistryValues	The RtlQueryRegistryValues routine allows the caller to query several values from the registry subtree with a single call.
RtlSanitizeUnicodeStringPadding	Sanitizes the padding for a Unicode string.
RtlSecureZeroMemory	The RtlSecureZeroMemory routine fills a block of memory with zeros in a way that is guaranteed to be secure.
RtlSetAllBits	The RtlSetAllBits routine sets all bits in a given bitmap variable.
RtlSetBit	The RtlSetBit routine sets the specified bit in a bitmap to one.
RtlSetBits	The RtlSetBits routine sets all bits in a given range of a given bitmap variable.
RtlSetDaclSecurityDescriptor	The RtlSetDaclSecurityDescriptor routine sets the DACL information of an absolute-format security descriptor, or if there is already a DACL present in the security descriptor, it is superseded.
RtlStringFromGUID	The RtlStringFromGUID routine converts a given GUID from binary format into a Unicode string.
RtlTestBit	The RtlTestBit routine returns the value of a bit in a bitmap.
RtlTimeFieldsToTime	The RtlTimeFieldsToTime routine converts TIME_FIELDS information to a system time value.
RtlTimeToTimeFields	The RtlTimeToTimeFields routine converts system time into a TIME_FIELDS structure.
RtlUlongByteSwap	The RtlUlongByteSwap routine reverses the ordering of the four bytes in a 32-bit unsigned integer value.
RtlUlonglongByteSwap	The RtlUlonglongByteSwap routine reverses the ordering of the eight bytes in a 64-bit unsigned integer value.
RtlUnicodeStringToAnsiSize	The RtlUnicodeStringToAnsiSize routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.
RtlUnicodeStringToAnsiString	The RtlUnicodeStringToAnsiString routine converts a given Unicode string into an ANSI string.
RtlUnicodeStringToInteger	The RtlUnicodeStringToInteger routine converts a Unicode string representation of a number to the equivalent integer value.
RtlUnicodeToUTF8N	The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.
RtlUpcaseUnicodeChar	The RtlUpcaseUnicodeChar routine converts the specified Unicode character to uppercase.

TITLE	DESCRIPTION
RtlUshortByteSwap	The RtlUshortByteSwap routine reverses the ordering of the two bytes in a 16-bit unsigned integer value.
RtlUTF8ToUnicodeN	The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.
RtlValidRelativeSecurityDescriptor	The RtlValidRelativeSecurityDescriptor routine checks the validity of a self-relative security descriptor.
RtlValidSecurityDescriptor	The RtlValidSecurityDescriptor routine checks a given security descriptor's validity.
RtlVerifyVersionInfo	The RtlVerifyVersionInfo routine compares a specified set of operating system version requirements to the corresponding attributes of the currently running version of the operating system.
RtlWriteRegistryValue	The RtlWriteRegistryValue routine writes caller-supplied data into the registry along the specified relative path at the given value name.
RtlxAnsiStringToUnicodeSize	The RtlxAnsiStringToUnicodeSize routine returns the number of bytes that are required for a null-terminated Unicode string that is equivalent to a specified ANSI string.
RtlxUnicodeStringToAnsiSize	The RtlxUnicodeStringToAnsiSize routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.
RtlZeroMemory	The RtlZeroMemory routine fills a block of memory with zeros, given a pointer to the block and the length, in bytes, to be filled.
SeAccessCheck	The SeAccessCheck routine determines whether the requested access rights can be granted to an object protected by a security descriptor and an object owner.
SeAssignSecurity	The SeAssignSecurity routine builds a self-relative security descriptor for a new object, given the security descriptor of its parent directory and any originally requested security for the object.
SeAssignSecurityEx	The SeAssignSecurityEx routine builds a self-relative security descriptor for a new object given the following optional parameters:_a security descriptor of the object's parent directory, an explicit security descriptor for the object, and the object type.
SeCaptureSubjectContext	The SeCaptureSubjectContext routine captures the security context of the calling thread for access validation and auditing.
SeDeassignSecurity	The SeDeassignSecurity routine deallocates the memory associated with a security descriptor that was assigned using SeAssignSecurity.

TITLE	DESCRIPTION
SeEtwWriteKMCveEvent	The SeEtwWriteKMCveEvent function is a tracing function for publishing events when an attempted security vulnerability exploit is detected in your kernel-mode drivers.
SeLockSubjectContext	The SeLockSubjectContext routine locks the primary and impersonation tokens of a captured subject context.
SeReleaseSubjectContext	The SeReleaseSubjectContext routine releases a subject security context captured by an earlier call to SeCaptureSubjectContext.
SeUnlockSubjectContext	The SeUnlockSubjectContext routine unlocks the tokens of a captured subject context that were locked by a call to SeLockSubjectContext.
SeValidSecurityDescriptor	The SeValidSecurityDescriptor routine returns whether a given security descriptor is structurally valid.
TmCommitComplete	The TmCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction.
TmCommitEnlistment	The TmCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
TmCommitTransaction	The TmCommitTransaction routine initiates a commit operation for a specified transaction.
TmCreateEnlistment	The TmCreateEnlistment routine creates a new enlistment object for a transaction.
TmDereferenceEnlistmentKey	The TmDereferenceEnlistmentKey routine decrements the reference count for the key of a specified enlistment object.
TmEnableCallbacks	The TmEnableCallbacks routine enables a callback routine that receives transaction notifications.
TmGetTransactionId	The TmGetTransactionId routine retrieves a transaction object's unit of work (UOW) identifier.
TmInitializeTransactionManager	The TmInitializeTransactionManager routine initializes a transaction manager object.
TmIsTransactionActive	The TmIsTransactionActive routine indicates whether a specified transaction is in its active state.
TmPrepareComplete	The TmPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
TmPrepareEnlistment	The TmPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
TmPrePrepareComplete	The TmPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

TITLE	DESCRIPTION
TmPrePrepareEnlistment	The TmPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
TmReadOnlyEnlistment	The TmReadOnlyEnlistment routine sets a specified enlistment to be read-only.
TmRecoverEnlistment	The TmRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.
TmRecoverResourceManager	The TmRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
TmRecoverTransactionManager	The TmRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
TmReferenceEnlistmentKey	The TmReferenceEnlistmentKey routine increments the reference count for the key of a specified enlistment object and retrieves the key.
TmRenameTransactionManager	The TmRenameTransactionManager routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.
TmRequestOutcomeEnlistment	The TmRequestOutcomeEnlistment routine asks KTM to try to provide an immediate outcome (commit or rollback) for the transaction that is associated with a specified enlistment.
TmRollbackComplete	The TmRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
TmRollbackEnlistment	The TmRollbackEnlistment routine rolls back a specified enlistment.
TmRollbackTransaction	The TmRollbackTransaction routine initiates a rollback operation for a specified transaction.
TmSinglePhaseReject	The TmSinglePhaseReject routine informs KTM that the calling resource manager will not support a single-phase commit operation for a specified enlistment.
vDbgPrintEx	The vDbgPrintEx routine sends a string to the kernel debugger if certain conditions are met.
vDbgPrintExWithPrefix	The vDbgPrintExWithPrefix routine sends a string to the kernel debugger if certain conditions that you specify are met. This routine can append a prefix to debugger output to help organize debugging results.

TITLE	DESCRIPTION
VslCreateSecureSection	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
VslDeleteSecureSection	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
WmiQueryTraceInformation	The WmiQueryTraceInformation routine returns information about a WMI event trace.
WmiTraceMessage	The WmiTraceMessage routine adds a message to the output log of a WPP software tracing session.
WmiTraceMessageVa	The WmiTraceMessageVa routine adds a message to the output log of a WPP software tracing session.
WRITE_PORT_BUFFER_UCHAR	The WRITE_PORT_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified port.
WRITE_PORT_BUFFER_ULONG	The WRITE_PORT_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified port address.
WRITE_PORT_BUFFER_USHORT	The WRITE_PORT_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified port address.
WRITE_PORT_UCHAR	The WRITE_PORT_UCHAR routine writes a byte to the specified port address.
WRITE_PORT_ULONG	The WRITE_PORT_ULONG routine writes a ULONG value to the specified port address.
WRITE_PORT_USHORT	The WRITE_PORT_USHORT routine writes a USHORT value to the specified port address.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG64	The WRITE_REGISTER_BUFFER_ULONG64 function writes a number of ULONG64 values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.

TITLE	DESCRIPTION
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_ULONG64	The WRITE_REGISTER_ULONG64 function writes a ULONG64 value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.
ZwClose	The ZwClose routine closes an object handle.
ZwCommitComplete	The ZwCommitComplete routine notifies KTM that the calling resource manager has finished committing a transaction's data.
ZwCommitEnlistment	The ZwCommitEnlistment routine initiates the commit operation for a specified enlistment's transaction.
ZwCommitTransaction	The ZwCommitTransaction routine initiates a commit operation for a specified transaction.
ZwCreateDirectoryObject	The ZwCreateDirectoryObject routine creates or opens an object-directory object.
ZwCreateEnlistment	The ZwCreateEnlistment routine creates a new enlistment object for a transaction.
ZwCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
ZwCreateKey	The ZwCreateKey routine creates a new registry key or opens an existing one.
ZwCreateKeyTransacted	The ZwCreateKeyTransacted routine creates a new registry key or opens an existing one, and it associates the key with a transaction.
ZwCreateResourceManager	The ZwCreateResourceManager routine creates a resource manager object.
ZwCreateSection	The ZwCreateSection routine creates a section object.
ZwCreateTransaction	The ZwCreateTransaction routine creates a transaction object.
ZwCreateTransactionManager	The ZwCreateTransactionManager routine creates a new transaction manager object.
ZwDeleteKey	The ZwDeleteKey routine deletes an open key from the registry.
ZwDeleteValueKey	The ZwDeleteValueKey routine deletes a value entry matching a name from an open key in the registry. If no such entry exists, an error is returned.

TITLE	DESCRIPTION
ZwEnumerateKey	The ZwEnumerateKey routine returns information about a subkey of an open registry key.
ZwEnumerateTransactionObject	The ZwEnumerateTransactionObject routine enumerates the KTM objects on a computer.
ZwEnumerateValueKey	The ZwEnumerateValueKey routine gets information about the value entries of an open key.
ZwFlushKey	The ZwFlushKey routine forces a registry key to be committed to disk.
ZwGetNotificationResourceManager	The ZwGetNotificationResourceManager routine retrieves the next transaction notification from a specified resource manager's notification queue.
ZwLoadDriver	The ZwLoadDriver routine loads a driver into the system.
ZwMakeTemporaryObject	The ZwMakeTemporaryObject routine changes the attributes of an object to make it temporary.
ZwMapViewOfSection	The ZwMapViewOfSection routine maps a view of a section into the virtual address space of a subject process.
ZwOpenEnlistment	The ZwOpenEnlistment routine obtains a handle to an existing enlistment object.
ZwOpenEvent	The ZwOpenEvent routine opens a handle to an existing named event object with the specified desired access.
ZwOpenFile	The ZwOpenFile routine opens an existing file, directory, device, or volume.
ZwOpenKey	The ZwOpenKey routine opens an existing registry key.
ZwOpenKeyEx	The ZwOpenKeyEx routine opens an existing registry key.
ZwOpenKeyTransacted	The ZwOpenKeyTransacted routine opens an existing registry key and associates the key with a transaction.
ZwOpenKeyTransactedEx	The ZwOpenKeyTransactedEx routine opens an existing registry key and associates the key with a transaction.
ZwOpenResourceManager	The ZwOpenResourceManager routine returns a handle to an existing resource manager object.
ZwOpenSection	The ZwOpenSection routine opens a handle for an existing section object.
ZwOpenSymbolicLinkObject	The ZwOpenSymbolicLinkObject routine opens an existing symbolic link.

TITLE	DESCRIPTION
ZwOpenTransaction	The ZwOpenTransaction routine obtains a handle to an existing transaction object.
ZwOpenTransactionManager	The ZwOpenTransactionManager routine obtains a handle to an existing transaction manager object.
ZwPrepareComplete	The ZwPrepareComplete routine notifies KTM that the calling resource manager has finished preparing a transaction's data.
ZwPrepareEnlistment	The ZwPrepareEnlistment routine initiates the prepare operation for a specified enlistment's transaction.
ZwPrePrepareComplete	The ZwPrePrepareComplete routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.
ZwPrePrepareEnlistment	The ZwPrePrepareEnlistment routine initiates the pre-prepare operation for a specified enlistment's transaction.
ZwQueryFullAttributesFile	The ZwQueryFullAttributesFile routine supplies network open information for the specified file.
ZwQueryInformationEnlistment	The ZwQueryInformationEnlistment routine retrieves information about a specified enlistment object.
ZwQueryInformationFile	The ZwQueryInformationFile routine returns various kinds of information about a file object.
ZwQueryInformationResourceManager	The ZwQueryInformationResourceManager routine retrieves information about a specified resource manager object.
ZwQueryInformationTransaction	The ZwQueryInformationTransaction routine retrieves information about a specified transaction.
ZwQueryInformationTransactionManager	The ZwQueryInformationTransactionManager routine retrieves information about a specified transaction manager object.
ZwQueryKey	The ZwQueryKey routine provides information about the class of a registry key, and the number and sizes of its subkeys.
ZwQuerySymbolicLinkObject	The ZwQuerySymbolicLinkObject routine returns a Unicode string that contains the target of a symbolic link.
ZwQueryValueKey	The ZwQueryValueKey routine returns a value entry for a registry key.
ZwReadFile	The ZwReadFile routine reads data from an open file.
ZwReadOnlyEnlistment	The ZwReadOnlyEnlistment routine sets a specified enlistment to be read-only.
ZwRecoverEnlistment	The ZwRecoverEnlistment routine initiates a recovery operation for the transaction that is associated with a specified enlistment.

TITLE	DESCRIPTION
ZwRecoverResourceManager	The ZwRecoverResourceManager routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.
ZwRecoverTransactionManager	The ZwRecoverTransactionManager routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.
ZwRollbackComplete	The ZwRollbackComplete routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.
ZwRollbackEnlistment	The ZwRollbackEnlistment routine rolls back the transaction that is associated with a specified enlistment.
ZwRollbackTransaction	The ZwRollbackTransaction routine initiates a rollback operation for a specified transaction.
ZwRollforwardTransactionManager	The ZwRollforwardTransactionManager routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.
ZwSetInformationEnlistment	The ZwSetInformationEnlistment routine sets information for a specified enlistment object.
ZwSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
ZwSetInformationResourceManager	The ZwSetInformationResourceManager routine is not used.
ZwSetInformationTransaction	The ZwSetInformationTransaction routine sets information for a specified transaction.
ZwSetValueKey	The ZwSetValueKey routine creates or replaces a registry key's value entry.
ZwSinglePhaseReject	The ZwSinglePhaseReject routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.
ZwUnloadDriver	The ZwUnloadDriver routine unloads a driver from the system.
ZwUnmapViewOfSection	The ZwUnmapViewOfSection routine unmaps a view of a section from the virtual address space of a subject process.
ZwWriteFile	The ZwWriteFile routine writes data to an open file.

Callback functions

TITLE	DESCRIPTION

TITLE	DESCRIPTION
ALLOCATE_FUNCTION_EX	The LookasideListAllocateEx routine allocates the storage for a new lookaside-list entry when a client requests an entry from a lookaside list that is empty.
BOUND_CALLBACK	The BoundCallback routine is executed whenever the system issues a bounds exception for a user-mode thread.
D3COLD_REQUEST_AUX_POWER	Enables the function device object (FDO) to convey its auxiliary power requirement.
D3COLD_REQUEST_CORE_POWER_RAIL	Enables the function device object (FDO) to indicate whether the core power rail is needed.
D3COLD_REQUEST_PERST_DELAY	Enables the function device object (FDO) to convey its requirement for a fixed delay time.
DMA_COMPLETION_ROUTINE	The DmaCompletionRoutine callback routine notifies the driver that previously requested a system DMA transfer that this transfer is complete.
DRIVER_ADD_DEVICE	The AddDevice routine is responsible for creating functional device objects (FDO) or filter device objects (filter DO) for devices enumerated by the Plug and Play (PnP) manager.
DRIVER_CANCEL	The Cancel routine cancels an I/O operation.
DRIVER_CONTROL	This routine starts a DMA data-transfer or a data transfer operation.
DRIVER_DISPATCH	The callback routine services various IRPs. For a list of function codes, see Remarks.
DRIVER_INITIALIZE	DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.
DRIVER_LIST_CONTROL	The AdapterListControl routine starts a direct memory access (DMA) scatter/gather operation.
DRIVER_STARTIO	The StartIo routine starts the I/O operation described by an IRP.
DRIVER_UNLOAD	The Unload routine performs any operations that are necessary before the system unloads the driver.
ENABLE_VIRTUALIZATION	The EnableVirtualization routine enables or disables virtualization for a PCI Express (PCIe) device that supports the single root I/O virtualization (SR-IOV) interface.
ETWENABLECALLBACK	The EtwEnableCallback function is an optional driver-supplied callback function that is used to receive enable or disable notifications.
EX_CALLBACK_FUNCTION	A filter driver's RegistryCallback routine can monitor, block, or modify a registry operation.

TITLE	DESCRIPTION
EXT_CALLBACK	An ExTimerCallback callback routine runs after an EX_TIMER timer object's time interval expires.
EXT_DELETE_CALLBACK	An ExTimerDeleteCallback callback routine runs when the operating system deletes an EX_TIMER timer object.
FPGA_BUS_SCAN	Reserved for future use. Triggers a bus scan at the parent of the FPGA device.
FPGA_CONTROL_CONFIG_SPACE	Reserved for future use. Enables or disables the access to the configuration space of the FPGA device.
FPGA_CONTROL_ERROR_REPORTING	Reserved for future use. Toggles the error reporting for the FPGA device and its parent bridge.
FPGA_CONTROL_LINK	Reserved for future use.
FREE_FUNCTION_EX	The LookasideListFreeEx routine frees the storage for a lookaside-list entry when a client tries to insert the entry into a lookaside list that is full.
GET_D3COLD_CAPABILITY	The GetBusDriverD3ColdSupport routine enables the driver for a device to query whether the enumerating bus driver supports the D3cold device power state.
GET_D3COLD_LAST_TRANSITION_STATUS	The GetLastTransitionStatus routine enables the driver for a device to query whether the most recent transition to the D3hot substate was followed by a transition to the D3cold substate.
GET_DMA_ADAPTER	The GetDmaAdapter routine returns a DMA_ADAPTER structure for the target device.
GET_IDLE_WAKE_INFO	The GetIdleWakeInfo routine enables the driver for a device to discover the device power states from which the device can signal a wake event.
GET_SDEV_IDENTIFIER	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
GET_SET_DEVICE_DATA	The GetBusData routine reads data from the device's configuration space.
GET_UPDATED_BUS_RESOURCE	Reports the latest resource lists.
GET_VIRTUAL_DEVICE_DATA	The GetVirtualFunctionData routine reads data from the PCI Express (PCIe) configuration space of a virtual function (VF) on a device that supports the single root I/O virtualization (SR-IOV) interface.
GET_VIRTUAL_DEVICE_LOCATION	The GetLocation routine returns the device location of a PCI Express (PCIe) virtual function (VF) on a PCI bus. A device that supports the single root I/O virtualization (SR-IOV) interface can expose one or more VFs on the PCI bus.

TITLE	DESCRIPTION
GET_VIRTUAL_DEVICE_RESOURCES	The GetResources routine returns the resources that the PCI Express (PCIe) physical function (PF) requires in order to enable virtualization on a device that supports the single root I/O virtualization (SR-IOV) interface.
GET_VIRTUAL_FUNCTION_PROBED_BARS	The GetVirtualFunctionProbedBars routine returns the values of the PCI Express (PCIe) Base Address Registers (BARs) of a device that supports the single root I/O virtualization (SR-IOV) interface.
IO_COMPLETION_ROUTINE	The IoCompletion routine completes the processing of I/O operations.
IO_CSQ_ACQUIRE_LOCK	The CsqAcquireLock routine is used by the system to acquire the lock for a driver-implemented, cancel-safe IRP queue.
IO_CSQ_COMPLETE_CANCELED_IRP	The CsqCompleteCanceledIrp routine is used by the system to signal to the driver that it can complete a canceled IRP.
IO_CSQ_INSERT_IRP	The CsqInsertIrp routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.
IO_CSQ_INSERT_IRP_EX	The CsqInsertIrpEx routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.
IO_CSQ_PEEK_NEXT_IRP	The CsqPeekNextIrp routine is used by the system to find the next matching IRP in a driver-implemented, cancel-safe IRP queue.
IO_CSQ_RELEASE_LOCK	The CsqReleaseLock routine is used by the system to release the lock that was acquired using CsqAcquireLock.
IO_CSQ_REMOVE_IRP	The CsqRemoveIrp routine is used by the system to remove the specified IRP from a driver-implemented, cancel-safe IRP queue.
IO_DPC_ROUTINE	The DpcForIsr routine finishes the servicing of an I/O operation, after an InterruptService routine returns.
IO_SESSION_NOTIFICATION_FUNCTION	The IO_SESSION_NOTIFICATION_FUNCTION function type defines a callback routine through which a driver receives notifications of changes in the state of user sessions that the driver is interested in.
IO_TIMER_ROUTINE	The IoTimer routine is a DPC that, if registered, is called once per second.
IO_WORKITEM_ROUTINE	A WorkItem routine performs the processing for a work item that was queued by the IoQueueWorkItem routine.

TITLE	DESCRIPTION
IO_WORKITEM_ROUTINE_EX	A WorkItemEx routine performs the processing for a work item that was queued by the IoQueueWorkItemEx or IoTryQueueWorkItem routine.
IOMMU_DEVICE_FAULT_HANDLER	Reports fault from a specific device and domain.
IOMMU_DOMAIN_ATTACH_DEVICE	Attaches a device to an existing domain.
IOMMU_DOMAIN_CONFIGURE	Configures a domain for use.
IOMMU_DOMAIN_CREATE	Creates a new DMA remapping device domain (a container for a set of page tables).
IOMMU_DOMAIN_DELETE	Deletes an existing domain.
IOMMU_DOMAIN_DETACH_DEVICE	Detaches a device from an existing domain.
IOMMU_FLUSH_DOMAIN	Flushes the TLB for all entries which match this domain.
IOMMU_FLUSH_DOMAIN_VA_LIST	Flushes the TLB for all entries that match the specified domain's ASID and one of the addresses in the provided list.
IOMMU_MAP_IDENTITY_RANGE	Creates an identity mapping for the provided MDL in the provided domain.
IOMMU_MAP_LOGICAL_RANGE	Maps a range of pages into the address space of a domain.
IOMMU_QUERY_INPUT_MAPPINGS	Attempts to find input mapping IDs which are valid for the given device and populate the provied buffer with those IDs.
IOMMU_SET_DEVICE_FAULT_REPORTING	This routine sets the device fault reporting state to a device already attached to a domain.
IOMMU_UNMAP_IDENTITY_RANGE	Deletes an identity mapping for the specified MDL.
IOMMU_UNMAP_LOGICAL_RANGE	Unmaps a linear range from a domain.
KBUGCHECK_CALLBACK_ROUTINE	The BugCheckCallback routine is executed whenever the system issues a bug check.
KBUGCHECK_REASON_CALLBACK_ROUTINE	Driver-implemented callback functions that the system executes when it issues a bug check.
KDEFERRED_ROUTINE	The callback routine performs actions, after an InterruptService returns, of a threaded DPC, The CustomDpc routine finishes the servicing of an I/O operation, after an InterruptService routine returns.The CustomThreadedDpc routine performs the action of a threaded DPC. The system executes this routine when the threaded DPC runs.The CustomTimerDpc routine executes after a timer object's time interval expires.
KIPI_BROADCAST_WORKER	The IpiGenericCall routine runs simultaneously on all processors.

TITLE	DESCRIPTION
KMESSAGE_SERVICE_ROUTINE	An InterruptMessageService routine services a message-signaled interrupt.
KSERVICE_ROUTINE	The InterruptService routine (ISR) quickly services a device interrupt and schedules post-interrupt processing of received data, if necessary.
KSTART_ROUTINE	The ThreadStart routine provides an entry point for a driver-created system thread.
KSYNCHRONIZE_ROUTINE	The SynchCritSection routine is used to access hardware resources or driver data that are shared with a driver's InterruptService routine.
MM_MDL_ROUTINE	A driver-supplied callback routine that is invoked after a memory descriptor list (MDL) is mapped by calling the MmMapMdl function.
PALLOCATE_ADAPTER_CHANNEL	The AllocateAdapterChannel routine prepares the system for a DMA operation on behalf of the target device object, and then calls the driver-supplied AdapterControl routine to carry out the DMA operation.
PALLOCATE_ADAPTER_CHANNEL_EX	The AllocateAdapterChannelEx routine allocates the resources that are needed to perform a DMA transfer, and then calls the driver-supplied AdapterControl routine to initiate the DMA transfer.
PALLOCATE_COMMON_BUFFER	The AllocateCommonBuffer routine allocates memory and maps it so that it is simultaneously accessible from both the processor and a device for DMA operations.
PALLOCATE_COMMON_BUFFER_EX	The AllocateCommonBufferEx routine allocates memory for a common buffer and maps this memory so that it can be accessed both by the processor and by a device that performs DMA operations.
PALLOCATE_COMMON_BUFFER_WITH_BOUNDS	
PALLOCATE_DOMAIN_COMMON_BUFFER	
PBUILD_MDL_FROM_SCATTER_GATHER_LIST	The BuildMdlFromScatterGatherList routine builds an MDL from a scatter/gather list allocated by the system.Note This routine is reserved for system use.
PBUILD_SCATTER_GATHER_LIST	The BuildScatterGatherList routine prepares the system for a DMA operation, using a driver-supplied buffer to build the scatter/gather list.
PBUILD_SCATTER_GATHER_LIST_EX	The BuildScatterGatherListEx routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.

TITLE	DESCRIPTION
PCALCULATE_SCATTER_GATHER_LIST_SIZE	The CalculateScatterGatherList routine calculates the size, in bytes, of scatter/gather list necessary to hold a given buffer.
PCANCEL_ADAPTER_CHANNEL	The CancelAdapterChannel routine tries to cancel a pending request to allocate a DMA channel.
PCANCEL_MAPPED_TRANSFER	The CancelMappedTransfer routine cancels a system DMA transfer that is currently mapped to an adapter.
PCI_MSIX_MASKUNMASK_ENTRY	The MaskTableEntry routine masks an interrupt in the MSI-X hardware interrupt table.
PCI_MSIX_SET_ENTRY	The SetTableEntry routine sets the message ID for a table entry in the MSI-X hardware interrupt table.
PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK	The ClfsAdvanceTailCallback function advances the base log sequence number (LSN) of the client's log.
PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK	The ClfsLogGrowthCompleteCallback function implements the actions that the client will take when space is freed in a log that had previously been full.
PCLFS_CLIENT_LOG_UNPINNED_CALLBACK	The ClfsLogUnpinnedCallback function implements any actions that the client will take when a log that had previously been pinned becomes unpinned.
PCONFIGURE_ADAPTER_CHANNEL	The ConfigureAdapterChannel routine invokes a custom function that is implemented by the DMA controller represented by an adapter object.
PCW_CALLBACK	Providers can implement a PCW_CALLBACK function to receive notification when consumers perform certain actions, such as adding or removing counters from a query.
PDEVICE_RESET_HANDLER	The DeviceReset routine is used to reset and recover a malfunctioning device.
PFLUSH_ADAPTER_BUFFERS	The FlushAdapterBuffers routine flushes any data remaining in the system DMA controller's internal cache or in a bus-master adapter's internal cache at the end of a DMA transfer operation.
PFLUSH_ADAPTER_BUFFERS_EX	The FlushAdapterBuffersEx routine flushes any data that remains in the data cache at the end of a DMA transfer operation performed by a system DMA controller or bus-master device.
PFLUSH_DMA_BUFFER	
PFREE_ADAPTER_CHANNEL	The FreeAdapterChannel routine releases the system DMA controller when a driver has completed all DMA operations necessary to satisfy the current IRP.
PFREE_ADAPTER_OBJECT	The FreeAdapterObject routine releases the specified adapter object after a driver has completed all DMA operations.

TITLE	DESCRIPTION
PFREE_COMMON_BUFFER	The FreeCommonBuffer routine frees a common buffer allocated by AllocateCommonBuffer, along with all resources the buffer uses.
PFREE_MAP_REGISTERS	The FreeMapRegisters routine releases a set of map registers that were saved from a call to AllocateAdapterChannel.
PGET_DMA_ADAPTER_INFO	The GetDmaAdapterInfo routine retrieves information about the hardware capabilities of a system DMA channel.
PGET_DMA_ALIGNMENT	The GetDmaAlignment routine returns the alignment requirements of the DMA system.
PGET_DMA_DOMAIN	
PGET_DMA_TRANSFER_INFO	The GetDmaTransferInfo routine calculates the allocation requirements for a scatter/gather DMA transfer.
PGET_SCATTER_GATHER_LIST	The GetScatterGatherList routine prepares the system for a DMA scatter/gather operation on behalf of the target device object, through either the system DMA controller or a bus-master adapter.
PGET_SCATTER_GATHER_LIST_EX	The GetScatterGatherListEx routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.
PINITIALIZE_DMA_TRANSFER_CONTEXT	The InitializeDmaTransferContext routine initializes an opaque DMA transfer context that is used to track pending allocations of DMA resources.
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.
PJOIN_DMA_DOMAIN	
PLEAVE_DMA_DOMAIN	
PMAP_TRANSFER	The MapTransfer routine sets up map registers for an adapter object to map a DMA transfer from a locked-down buffer.
PMAP_TRANSFER_EX	The MapTransferEx routine sets up map registers to map the physical addresses in a scatter/gather list to the logical addresses that are required to do a DMA transfer.

TITLE	DESCRIPTION
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK	The ComponentActiveConditionCallback callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK	The ComponentCriticalTransitionCallback callback routine handles a transition of the specified component between the F0 (fully on) and low-power Fx component power states.
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK	The ComponentIdleConditionCallback callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
PO_FX_COMPONENT_IDLE_STATE_CALLBACK	The ComponentIdleStateCallback callback routine notifies the driver of a pending change to the Fx power state of the specified component.
PO_FX_COMPONENT_PERF_STATE_CALLBACK	The ComponentPerfStateCallback callback routine notifies the driver that its request to change the performance state of a component is complete.
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK	The DevicePowerNotRequiredCallback callback routine notifies the device driver that the device is not required to stay in the D0 power state.
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK	The DevicePowerRequiredCallback callback routine notifies the device driver that the device must enter and remain in the D0 power state.
PO_FX_POWER_CONTROL_CALLBACK	The PowerControlCallback callback routine performs a power control operation that is requested by the power management framework (PoFx).
POB_POST_OPERATION_CALLBACK	The ObjectPostCallback routine is called by the operating system after a process or thread handle operation occurs.
POB_PRE_OPERATION_CALLBACK	The ObjectPreCallback routine is called by the operating system when a process or thread handle operation occurs.
PPUT_DMA_ADAPTER	The PutDmaAdapter routine frees a DMA_ADAPTER structure previously allocated by IoGetDmaAdapter.
PPUT_SCATTER_GATHER_LIST	The PutScatterGatherList routine frees the previously allocated map registers and scatter/gather list used in scatter/gather DMA.
PREAD_DMA_COUNTER	The ReadDmaCounter routine returns the number of bytes remaining to be transferred during the current subordinate DMA operation.
PREENUMERATE_SELF	A ReenumerateSelf routine requests that a bus driver reenumerate a child device.
PROCESSOR_HALT_ROUTINE	A Halt callback routine transitions the processor to an idle state.

TITLE	DESCRIPTION
PTM_RM_NOTIFICATION	A resource manager's ResourceManagerNotification callback routine receives and handles transaction notifications.
REQUEST_POWER_COMPLETE	The PowerCompletion callback routine completes the processing of a power IRP.
RTL_QUERY_REGISTRY_ROUTINE	The QueryRoutine routine provides information about a registry value that was requested in a preceding call to the RtlQueryRegistryValues routine.
SET_D3COLD_SUPPORT	The SetD3ColdSupport routine enables or disables transitions to the D3cold device power state.
SET_VIRTUAL_DEVICE_DATA	The SetVirtualFunctionData routine writes data to the PCI Express (PCIe) configuration space of a virtual function (VF) on a device that supports the single root I/O virtualization (SR-IOV) interface.
TRANSLATE_BUS_ADDRESS	The TranslateBusAddress routine translates addresses on the parent bus to logical addresses.

Structures

TITLE	DESCRIPTION
DMA_ADAPTER	The DMA_ADAPTER structure describes a system-defined interface to a DMA controller for a given device. A driver calls IoGetDmaAdapter to obtain this structure.
DMA_OPERATIONS	The DMA_OPERATIONS structure provides a table of pointers to functions that control the operation of a DMA controller.
ACCESS_STATE	The ACCESS_STATE structure describes the state of an access in progress.
ACL	The ACL structure is the header of an access-control list (ACL). A complete ACL consists of an ACL structure followed by an ordered list of zero or more access-control entries (ACE).
BATTERY_REPORTING_SCALE	Battery miniclass drivers fill in this structure in response to certain BatteryMiniQueryInformation requests.
BOOTDISK_INFORMATION	The BOOTDISK_INFORMATION structure contains basic information describing the boot and system disks.
BOOTDISK_INFORMATION_EX	The BOOTDISK_INFORMATION_EX structure contains extended information describing the boot and system disks.
BUS_INTERFACE_STANDARD	The BUS_INTERFACE_STANDARD interface structure enables device drivers to make direct calls to parent bus driver routines. This structure defines the GUID_BUS_INTERFACE_STANDARD interface.

TITLE	DESCRIPTION
BUS_RESOURCE_UPDATE_INTERFACE	Enables device drivers to make direct calls to parent bus driver routines. This structure defines the GUID_BUS_RESOURCE_UPDATE_INTERFACE interface.
CLFS_LOG_NAME_INFORMATION	The CLFS_LOG_NAME_INFORMATION structure holds the name of a Common Log File System (CLFS) stream or log.
CLFS_MGMT_CLIENT_REGISTRATION	The CLFS_MGMT_CLIENT_REGISTRATION structure is given to CLFS management by clients who manage their own logs.
CLFS_MGMT_POLICY	The CLFS_MGMT_POLICY structure holds a description of a policy for managing a CLFS log.
CLFS_STREAM_ID_INFORMATION	The CLFS_STREAM_ID_INFORMATION structure holds a value that identifies a stream in a Common Log File System (CLFS) log.
CLS_CONTAINER_INFORMATION	The CLFS_CONTAINER_INFORMATION structure holds descriptive information for an individual container in a Common Log File System (CLFS) log.
CLS_INFORMATION	The CLFS_INFORMATION structure holds metadata and state information for a Common Log File System (CLFS) stream and/or its underlying physical log.
CLS_IO_STATISTICS	The CLFS_IO_STATISTICS structure holds I/O statistics data for a Common Log File System (CLFS) log.
CLS_IO_STATISTICS_HEADER	The CLFS_IO_STATISTICS_HEADER structure holds the header portion of a CLFS_IO_STATISTICS structure.
CLS_LSN	The CLFS_LSN structure identifies an individual record in a Common Log File System (CLFS) stream.
CLS_SCAN_CONTEXT	The CLFS_SCAN_CONTEXT structure holds context information to support a scan of the containers in a Common Log File System (CLFS) log.
CLS_WRITE_ENTRY	The CLFS_WRITE_ENTRY structure holds the address and size of a buffer that contains one unit of data to be written to a Common Log File System (CLFS) stream.
CM_EISA_FUNCTION_INFORMATION	The CM_EISA_FUNCTION_INFORMATION structure defines detailed EISA configuration information returned by HalGetBusData for the input BusDataType EisaConfiguration, or by HalGetBusDataByOffset for the input BusDataType EisaConfiguration and the Offset zero, assuming the caller-allocated Buffer is of sufficient Length.
CM_EISA_SLOT_INFORMATION	The CM_EISA_SLOT_INFORMATION structure defines EISA configuration header information returned by HalGetBusData for the input BusDataType = EisaConfiguration, or by HalGetBusDataByOffset for the inputs BusDataType = EisaConfiguration and Offset = 0, assuming the caller-allocated Buffer is of sufficient Length.

TITLE	DESCRIPTION
CM_FLOPPY_DEVICE_DATA	The CM_FLOPPY_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a floppy controller if the system can collect this information during the boot process.
CM_FULL_RESOURCE_DESCRIPTOR	The CM_FULL_RESOURCE_DESCRIPTOR structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.
CM_INT13_DRIVE_PARAMETER	The CM_INT13_DRIVE_PARAMETER structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a disk controller if the system can collect this information during the boot process.
CM_KEYBOARD_DEVICE_DATA	The CM_KEYBOARD_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a keyboard peripheral if the system can collect this information during the boot process.
CM_MCA_POS_DATA	The CM_MCA_POS_DATA structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.
CM_PARTIAL_RESOURCE_DESCRIPTOR	The CM_PARTIAL_RESOURCE_DESCRIPTOR structure specifies one or more system hardware resources, of a single type, assigned to a device.
CM_PARTIAL_RESOURCE_LIST	The CM_PARTIAL_RESOURCE_LIST structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.
CM_POWER_DATA	The CM_POWER_DATA structure contains information about a device's power management state and capabilities.
CM_RESOURCE_LIST	The CM_RESOURCE_LIST structure specifies all of the system hardware resources assigned to a device.
CM_SCSI_DEVICE_DATA	The CM_SCSI_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a SCSI HBA if the system can collect this information during the boot process.
CM_SERIAL_DEVICE_DATA	The CM_SERIAL_DEVICE_DATA structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a serial controller if the system can collect this information during the boot process.
COUNTED_REASON_CONTEXT	The COUNTED_REASON_CONTEXT structure contains one or more strings that give reasons for a power request.

TITLE	DESCRIPTION
D3COLD_AUX_POWER_AND_TIMING_INTERFACE	Allows device drivers to negotiate a higher auxiliary power for their PCI devices while in D3Cold state.
D3COLD_SUPPORT_INTERFACE	The D3COLD_SUPPORT_INTERFACE interface structure contains pointers to the routines in the GUID_D3COLD_SUPPORT_INTERFACE driver interface.
DEVICE_CAPABILITIES	A DEVICE_CAPABILITIES structure describes PnP and power capabilities of a device. This structure is returned in response to an IRP_MN_QUERY_CAPABILITIES IRP.
DEVICE_DESCRIPTION	The DEVICE_DESCRIPTION structure describes the attributes of the physical device for which a driver is requesting a DMA adapter.
DEVICE_FAULT_CONFIGURATION	This structure is used in seting the device fault reporting state
DEVICE_INTERFACE_CHANGE_NOTIFICATION	The DEVICE_INTERFACE_CHANGE_NOTIFICATION structure describes a device interface that has been enabled (arrived) or disabled (removed).
DEVICE_OBJECT	A device object represents a logical, virtual, or physical device for which a driver handles I/O requests.
DEVICE_RESET_INTERFACE_STANDARD	The DEVICE_RESET_INTERFACE_STANDARD structure enables function drivers to reset and recover malfunctioning devices. This structure describes the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
DMA_ADAPTER_INFO	The DMA_ADAPTER_INFO structure is a container for a DMA_ADAPTER_INFO_XXX structure that describes the capabilities of a system DMA controller.
DMA_ADAPTER_INFO_V1	The DMA_ADAPTER_INFO_V1 structure describes the capabilities of the system DMA controller that is represented by an adapter object.
DMA_IOMMU_INTERFACE	
DMA_TRANSFER_INFO	The DMA_TRANSFER_INFO structure is a container for a DMA_TRANSFER_INFO_XXX structure that describes the allocation requirements for a scatter/gather list.
DMA_TRANSFER_INFO_V1	The DMA_TRANSFER_INFO_V1 structure contains the allocation requirements for a scatter/gather list that describes the I/O data buffer for a DMA transfer.
DMA_TRANSFER_INFO_V2	Contains the allocation requirements for a scatter/gather list that describes the I/O data buffer for a DMA transfer.
DOMAIN_CONFIGURATION	Contains information required to configure a domain.
DOMAIN_CONFIGURATION_ARM64	Contains information required to configure a domain for an ARM64 system.

TITLE	DESCRIPTION
DRIVER_OBJECT	Each driver object represents the image of a loaded kernel-mode driver.
ENLISTMENT_BASIC_INFORMATION	The ENLISTMENT_BASIC_INFORMATION structure contains information about an enlistment object.
EXT_DELETE_PARAMETERS	The EXT_DELETE_PARAMETERS structure contains an extended set of parameters for the ExDeleteTimer routine.
EXT_SET_PARAMETERS	The EXT_SET_PARAMETERS structure contains an extended set of parameters for the ExSetTimer routine.
FAST_IO_DISPATCH	Contains a set of callback routines that a file system driver or file system filter driver (legacy) provides for fast I/O processing.
FAULT_INFORMATION	Contains information to set the fault report on a device.
FAULT_INFORMATION_ARM64	Contains fault information for an ARM64 system.
FAULT_INFORMATION_ARM64_FLAGS	Defines fault information.
FILE_BASIC_INFORMATION	The FILE_BASIC_INFORMATION structure is used as an argument to routines that query or set file information.
FILE_FS_DEVICE_INFORMATION	The FILE_FS_DEVICE_INFORMATION structure provides file system device information about the type of device object associated with a file object.
FILE_FULL_EA_INFORMATION	The FILE_FULL_EA_INFORMATION structure provides extended attribute (EA) information. This structure is used primarily by network drivers.
FILE_IO_PRIORITY_HINT_INFORMATION	The FILE_IO_PRIORITY_HINT_INFORMATION structure is used by the ZwQueryInformationFile and ZwSetInformationFile routines to query and set the default IRP priority hint for requests on the specified file handle.
FILE_IS_REMOTE_DEVICE_INFORMATION	The FILE_IS_REMOTE_DEVICE_INFORMATION structure is used as an argument to the ZwQueryInformationFile routine.
FILE_MEMORY_PARTITION_INFORMATION	Stores information about memory partition. This structure is used by the ZwSetInformationFile function.
FILE_NETWORK_OPEN_INFORMATION	The FILE_NETWORK_OPEN_INFORMATION structure is used as an argument to ZwQueryInformationFile.
FILE_OBJECT	The FILE_OBJECT structure is used by the system to represent a file object.
FILE_POSITION_INFORMATION	The FILE_POSITION_INFORMATION structure is used as an argument to routines that query or set file information.

TITLE	DESCRIPTION
FILE_STANDARD_INFORMATION	The FILE_STANDARD_INFORMATION structure is used as an argument to routines that query or set file information.
FILE_STANDARD_INFORMATION_EX	The FILE_STANDARD_INFORMATION_EX structure is used as an argument to routines that query or set file information.
FPGA_CONTROL_INTERFACE	Reserved for future use.
FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS	The FUNCTION_LEVEL_DEVICE_RESET_PARAMETER structure is used as an argument to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
GENERIC_MAPPING	The GENERIC_MAPPING structure describes the ACCESS_MASK value of specific access rights associated with each type of generic access right.
HWPROFILE_CHANGE_NOTIFICATION	The HWPROFILE_CHANGE_NOTIFICATION structure describes an event related to a hardware profile configuration change.
IMAGE_POLICY_ENTRY	This structure is not supported.
IMAGE_POLICY_METADATA	This structure is not supported.
INPUT_MAPPING_ELEMENT	Contains the input mapping IDs for a device.
INTERFACE	The INTERFACE structure describes an interface that is exported by a driver for use by other drivers.
IO_CONNECT_INTERRUPT_PARAMETERS	The IO_CONNECT_INTERRUPT_PARAMETERS structure contains the parameters that a driver supplies to the IoConnectInterruptEx routine to register an interrupt service routine (ISR).
IO_DISCONNECT_INTERRUPT_PARAMETERS	The IO_DISCONNECT_INTERRUPT_PARAMETERS structure describes the parameters when unregistering an interrupt-handling routine with IoDisconnectInterruptEx.
IO_ERROR_LOG_PACKET	The IO_ERROR_LOG_PACKET structure serves as the header for an error log entry.
IO_INTERRUPT_MESSAGE_INFO	The IO_INTERRUPT_MESSAGE_INFO structure describes the driver's message-signaled interrupts.
IO_INTERRUPT_MESSAGE_INFO_ENTRY	The IO_INTERRUPT_MESSAGE_INFO_ENTRY structure describes the properties of a single message-signaled interrupt.
IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS	The IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure contains the connection context for a registered interrupt service routine (ISR) that was connected to an interrupt or interrupts by a previous call to the IoConnectInterruptEx routine.

TITLE	DESCRIPTION
IO_RESOURCE_DESCRIPTOR	The IO_RESOURCE_DESCRIPTOR structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of IO_RESOURCE_DESCRIPTOR structures is contained within each IO_RESOURCE_LIST structure.
IO_RESOURCE_LIST	The IO_RESOURCE_LIST structure describes a range of raw hardware resources, of various types, that can be used by a device.
IO_RESOURCE_REQUIREMENTS_LIST	The IO_RESOURCE_REQUIREMENTS_LIST structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.
IO_SECURITY_CONTEXT	The IO_SECURITY_CONTEXT structure represents the security context of an IRP_MJ_CREATE request.
IO_SESSION_CONNECT_INFO	The IO_SESSION_CONNECT_INFO structure provides information about a user session.
IO_SESSION_STATE_INFORMATION	The IO_SESSION_STATE_INFORMATION structure contains information about the state of a user session.
IO_SESSION_STATE_NOTIFICATION	The IO_SESSION_STATE_NOTIFICATION structure contains information that a kernel-mode driver supplies to the IoRegisterContainerNotification routine when the driver registers to receive notifications of session events.
IO_STACK_LOCATION	The IO_STACK_LOCATION structure defines an I/O stack location, which is an entry in the I/O stack that is associated with each IRP.
IO_STATUS_BLOCK	A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.
IRP	The IRP structure is a partially opaque structure that represents an I/O request packet. Drivers can use the following members of the IRP structure.
KBUGCHECK_ADD_PAGES	The KBUGCHECK_ADD_PAGES structure describes one or more pages of driver-supplied data to be written by a BugCheckAddPagesCallback callback routine to the crash dump file.
KBUGCHECK_DUMP_IO	The KBUGCHECK_DUMP_IO structure describes an I/O operation on the crash dump file.
KBUGCHECK_SECONDARY_DUMP_DATA	The KBUGCHECK_SECONDARY_DUMP_DATA structure describes a section of driver-supplied data to be written by BugCheckSecondaryDumpDataCallback to the crash dump file.
KDPC_WATCHDOG_INFORMATION	The KDPC_WATCHDOG_INFORMATION structure holds time-out information about the current deferred procedure call (DPC).

TITLE	DESCRIPTION
KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT	The KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure describes the notification context that is passed to a registered callback function when a new processor is dynamically added to a hardware partition.
KEY_BASIC_INFORMATION	The KEY_BASIC_INFORMATION structure defines a subset of the full information that is available for a registry key.
KEY_FULL_INFORMATION	The KEY_FULL_INFORMATION structure defines the information available for a registry key, including information about its subkeys and the maximum length for their names and value entries.
KEY_NODE_INFORMATION	The KEY_NODE_INFORMATION structure defines the basic information available for a registry (sub)key.
KEY_VALUE_BASIC_INFORMATION	The KEY_VALUE_BASIC_INFORMATION structure defines a subset of the full information available for a value entry of a registry key.
KEY_VALUE_ENTRY	The KEY_VALUE_ENTRY structure is used by the REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure to describe a single value entry for a registry key.
KEY_VALUE_FULL_INFORMATION	The KEY_VALUE_FULL_INFORMATION structure defines information available for a value entry of a registry key.
KEY_VALUE_PARTIAL_INFORMATION	The KEY_VALUE_PARTIAL_INFORMATION structure defines a subset of the value information available for a value entry of a registry key.
KEY_WRITE_TIME_INFORMATION	The KEY_WRITE_TIME_INFORMATION structure is used by the system to set the last write time for a registry key.
KTMOBJECT_CURSOR	The KTMOBJECT_CURSOR structure receives enumeration information about KTM objects when a component calls ZwEnumerateTransactionObject.
KTRIAGE_DUMP_DATA_ARRAY	
LINK_SHARE_ACCESS	The share access structure used by file systems for only link files.
LUID_AND_ATTRIBUTES	LUID_AND_ATTRIBUTES represents a locally unique identifier (LUID) and its attributes.
MDL	An MDL structure is a partially opaque structure that represents a memory descriptor list (MDL).
MEM_EXTENDED_PARAMETER	
MM_PHYSICAL_ADDRESS_LIST	The MM_PHYSICAL_ADDRESS_LIST structure specifies a range of physical addresses.

TITLE	DESCRIPTION
OB_CALLBACK_REGISTRATION	The OB_CALLBACK_REGISTRATION structure specifies the parameters when the ObRegisterCallbacks routine registers ObjectPreCallback and ObjectPostCallback callback routines.
OB_OPERATION_REGISTRATION	The OB_OPERATION_REGISTRATION structure specifies ObjectPreCallback and ObjectPostCallback callback routines and the types of operations that the routines are called for.
OB_POST_CREATE_HANDLE_INFORMATION	The OB_POST_CREATE_HANDLE_INFORMATION structure provides information to a ObjectPostCallback routine about a thread or process handle that has been opened.
OB_POST_DUPLICATE_HANDLE_INFORMATION	The OB_POST_DUPLICATE_HANDLE_INFORMATION structure provides information to an ObjectPostCallback routine about a thread or process handle that has been duplicated.
OB_POST_OPERATION_INFORMATION	The OB_POST_OPERATION_INFORMATION structure provides information about a process or thread handle operation to an ObjectPostCallback routine.
OB_POST_OPERATION_PARAMETERS	The OB_POST_OPERATION_PARAMETERS union describes the operation-specific parameters for an ObjectPostCallback routine.
OB_PRE_CREATE_HANDLE_INFORMATION	The OB_PRE_CREATE_HANDLE_INFORMATION structure provides information to an ObjectPreCallback routine about a thread or process handle that is being opened.
OB_PRE_DUPLICATE_HANDLE_INFORMATION	The OB_PRE_DUPLICATE_HANDLE_INFORMATION structure provides information to an ObjectPreCallback routine about a thread or process handle that is being duplicated.
OB_PRE_OPERATION_INFORMATION	The OB_PRE_OPERATION_INFORMATION structure provides information about a process or thread handle operation to an ObjectPreCallback routine.
OB_PRE_OPERATION_PARAMETERS	The OB_PRE_OPERATION_PARAMETERS union describes the operation-specific parameters for an ObjectPreCallback routine.
OSVERSIONINFOEXW	The RTL_OSVERSIONINFOEXW structure contains operating system version information.
OSVERSIONINFOW	The RTL_OSVERSIONINFOW structure contains operating system version information.
PCI_CAPABILITIES_HEADER	The PCI_CAPABILITIES_HEADER structure defines a header that is present in every PCI capability structure.
PCI_COMMON_CONFIG	The PCI_COMMON_CONFIG structure is obsolete.
PCI_DEVICE_PRESENT_INTERFACE	The PCI_DEVICE_PRESENT_INTERFACE structure is reserved for system use.

TITLE	DESCRIPTION
PCI_EXPRESS_AER_CAPABILITIES	The PCI_EXPRESS_AER_CAPABILITIES structure describes a PCI Express (PCIe) advanced error capabilities and control register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_AER_CAPABILITY	The PCI_EXPRESS_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure.
PCI_EXPRESS_BRIDGE_AER_CAPABILITY	The PCI_EXPRESS_BRIDGE_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure for a PCIe bridge device.
PCI_EXPRESS_CORRECTABLE_ERROR_MASK	The PCI_EXPRESS_CORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) correctable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_CORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_CORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) correctable error status register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_ENHANCED_CAPABILITY_HEADER	The PCI_EXPRESS_ENHANCED_CAPABILITY_HEADER structure describes the header for a PCI Express (PCIe) extended capability structure.
PCI_EXPRESS_ERROR_SOURCE_ID	The PCI_EXPRESS_ERROR_SOURCE_ID structure describes the identifiers of the first correctable error and the first uncorrectable error that are reported in the PCI Express (PCIe) root error status register.
PCI_EXPRESS_LINK_QUIESCENT_INTERFACE	The PCI_EXPRESS_LINK_QUIESCENT_INTERFACE structure is reserved for system use.
PCI_EXPRESS_PTM_CAPABILITY	Reserved. Do not use.
PCI_EXPRESS_PTM_CAPABILITY_REGISTER	Reserved. Do not use.
PCI_EXPRESS_PTM_CONTROL_REGISTER	Reserved. Do not use.
PCI_EXPRESS_ROOT_ERROR_COMMAND	The PCI_EXPRESS_ROOT_ERROR_COMMAND structure describes a PCI Express (PCIe) root error command register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_ROOT_ERROR_STATUS	The PCI_EXPRESS_ROOT_ERROR_STATUS structure describes a PCI Express (PCIe) root error status register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_ROOT_PORT_INTERFACE	The PCI_EXPRESS_ROOT_PORT_INTERFACE structure is reserved for system use.
PCI_EXPRESS_ROOTPORT_AER_CAPABILITY	The PCI_EXPRESS_ROOTPORT_AER_CAPABILITY structure describes a PCI Express (PCIe) advanced error reporting capability structure for a root port or a root complex event collector.

TITLE	DESCRIPTION
PCI_EXPRESS_SEC_AER_CAPABILITIES	The PCI_EXPRESS_SEC_AER_CAPABILITIES structure describes a PCI Express (PCIe) secondary error capabilities and control register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_MASK	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) secondary uncorrectable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_SEVERITY	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_SEVERITY structure describes a PCI Express (PCIe) secondary uncorrectable error severity register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_SEC_UNCORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) secondary uncorrectable error status register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_SERIAL_NUMBER_CAPABILITY	The PCI_EXPRESS_SERIAL_NUMBER_CAPABILITY structure describes a serial number for a PCI Express (PCIe) device.
PCI_EXPRESS_UNCORRECTABLE_ERROR_MASK	The PCI_EXPRESS_UNCORRECTABLE_ERROR_MASK structure describes a PCI Express (PCIe) uncorrectable error mask register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_UNCORRECTABLE_ERROR_SEVERITY	The PCI_EXPRESS_UNCORRECTABLE_ERROR_SEVERITY structure describes a PCI Express (PCIe) uncorrectable error severity register of a PCIe advanced error reporting capability structure.
PCI_EXPRESS_UNCORRECTABLE_ERROR_STATUS	The PCI_EXPRESS_UNCORRECTABLE_ERROR_STATUS structure describes a PCI Express (PCIe) uncorrectable error status register of a PCIe advanced error reporting capability structure.
PCI_MSIX_TABLE_CONFIG_INTERFACE	The PCI_MSIX_TABLE_CONFIG_INTERFACE structure enables device drivers to modify their MSI-X interrupt settings. This structure describes the GUID_MSIX_TABLE_CONFIG_INTERFACE interface.
PCI_PM_CAPABILITY	The PCI_PM_CAPABILITY structure reports the power management capabilities of the device.
PCI_PMC	The PCI_PMC structure is used to report the contents of the power management capabilities register.
PCI_PMCSR	The PCI_PMCSR structure is used to report the contents of the device's power management control status register.
PCI_PMCSR_BSE	The PCI_PMCSR_BSE structure is used to report the contents of the power management control status register for PCI bridge support extensions.

TITLE	DESCRIPTION
PCI_SLOT_NUMBER	The PCI_SLOT_NUMBER structure is obsolete.
PCI_X_CAPABILITY	The PCI_X_CAPABILITY structure reports the contents of the command and status registers of a device that is compliant with the PCI-X Addendum to the PCI Local Bus Specification.
PCW_CALLBACK_INFORMATION	The PCW_CALLBACK_INFORMATION union supplies details about the notification to send. A provider passes a pointer to this union as a parameter to the PcwCallback function.
PCW_COUNTER_DESCRIPTOR	The PCW_COUNTER_DESCRIPTOR structure supplies details about the notification to send.
PCW_COUNTER_INFORMATION	The PCW_COUNTER_INFORMATION structure describes attributes that identify a specific instance of a counter set.
PCW_DATA	The PCW_DATA structure describes the array of data blocks that are associated with an instance.
PCW_MASK_INFORMATION	The PCW_MASK_INFORMATION structure supplies details about the notification to send to the provider. This information is passed as part of the Info parameter to the PcwCallback function. This mask information is included in PCW_CALLBACK_INFORMATION.
PCW_REGISTRATION_INFORMATION	The PCW_REGISTRATION_INFORMATION structure supplies details about the provider and the counter set.
PLUGPLAY_NOTIFICATION_HEADER	A PLUGPLAY_NOTIFICATION_HEADER structure is included at the beginning of each PnP notification structure, such as a DEVICE_INTERFACE_CHANGE_NOTIFICATION structure.
PNP_BUS_INFORMATION	The PNP_BUS_INFORMATION structure describes a bus.
PO_FX_COMPONENT_IDLE_STATE	The PO_FX_COMPONENT_IDLE_STATE structure specifies the attributes of an Fx power state of a component in a device.
PO_FX_COMPONENT_PERF_INFO	The PO_FX_COMPONENT_PERF_INFO structure describes all the sets of performance states for a single component within a device.
PO_FX_COMPONENT_PERF_SET	The PO_FX_COMPONENT_PERF_SET structure represents a set of performance states for a single component within a device.
PO_FX_COMPONENT_V1	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_V2	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_DEVICE_V1	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).

TITLE	DESCRIPTION
PO_FX_DEVICE_V2	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_PERF_STATE	The PO_FX_PERF_STATE structure represents a performance state for a single component within a device.
PO_FX_PERF_STATE_CHANGE	The PO_FX_PERF_STATE_CHANGE structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.
POWER_PLATFORM_INFORMATION	The POWER_PLATFORM_INFORMATION structure contains information about the power capabilities of the system.
POWER_STATE	The POWER_STATE union specifies a system power state value or a device power state value.
PRIVILEGE_SET	The PRIVILEGE_SET structure specifies a set of security privileges. For more information, see the reference page for PRIVILEGE_SET in the Microsoft Windows SDK documentation.
PTM_CONTROL_INTERFACE	Resereved. Do not use.
REENUMERATE_SELF_INTERFACE_STANDARD	The REENUMERATE_SELF_INTERFACE_STANDARD interface structure enables a driver to request that its parent bus driver reenumerate the driver's device. This structure defines the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface.
REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION	The REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION structure contains information that a driver's RegistryCallback routine can use to free resources that the driver previously allocated for the context that is associated with a registry object.
REG_CREATE_KEY_INFORMATION	The REG_CREATE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key that is being created.
REG_CREATE_KEY_INFORMATION_V1	The REG_CREATE_KEY_INFORMATION_V1 structure contains information that a filter driver's RegistryCallback routine can use when a registry key is being created.
REG_DELETE_KEY_INFORMATION	The REG_DELETE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key is being deleted.
REG_DELETE_VALUE_KEY_INFORMATION	The REG_DELETE_VALUE_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry key's value is being deleted.
REG_ENUMERATE_KEY_INFORMATION	The REG_ENUMERATE_KEY_INFORMATION structure describes one subkey of a key whose subkeys are being enumerated.

TITLE	DESCRIPTION
REG_ENUMERATE_VALUE_KEY_INFORMATION	The REG_ENUMERATE_VALUE_KEY_INFORMATION structure describes one value entry of a key whose value entries are being enumerated.
REG_KEY_HANDLE_CLOSE_INFORMATION	The REG_KEY_HANDLE_CLOSE_INFORMATION structure contains information about a registry key whose handle is about to be closed.
REG_LOAD_KEY_INFORMATION	The REG_LOAD_KEY_INFORMATION structure contains information about a registry hive that is being loaded.
REG_POST_CREATE_KEY_INFORMATION	The REG_POST_CREATE_KEY_INFORMATION structure contains the result of an attempt to create a registry key.
REG_POST_OPERATION_INFORMATION	The REG_POST_OPERATION_INFORMATION structure contains information about a completed registry operation that a RegistryCallback routine can use.
REG_PRE_CREATE_KEY_INFORMATION	The REG_PRE_OPEN_KEY_INFORMATION structure contains the name of a registry key that is about to be opened.
REG_QUERY_KEY_INFORMATION	The REG_QUERY_KEY_INFORMATION structure describes the metadata that is about to be queried for a key.
REG_QUERY_KEY_NAME	The REG_QUERY_KEY_NAME structure describes the full registry key name of an object being queried.
REG_QUERY_KEY_SECURITY_INFORMATION	The REG_QUERY_KEY_SECURITY_INFORMATION structure receives security information for a registry key object.
REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION	The REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure describes the multiple value entries that are being retrieved for a key.
REG_QUERY_VALUE_KEY_INFORMATION	The REG_QUERY_VALUE_KEY_INFORMATION structure contains information about a registry key's value entry that is being queried.
REG_RENAME_KEY_INFORMATION	The REG_RENAME_KEY_INFORMATION structure contains the new name for a registry key whose name is about to be changed.
REG_REPLACE_KEY_INFORMATION	The REG_REPLACE_KEY_INFORMATION structure describes the metadata that is about to be replaced for a key.
REG_RESTORE_KEY_INFORMATION	The REG_RESTORE_KEY_INFORMATION structure contains the information for a registry key that is about to be restored.
REG_SAVE_KEY_INFORMATION	The REG_SAVE_KEY_INFORMATION structure contains the information for a registry key that is about to be saved.
REG_SET_INFORMATION_KEY_INFORMATION	The REG_SET_INFORMATION_KEY_INFORMATION structure describes a new setting for a key's metadata.

TITLE	DESCRIPTION
REG_SET_KEY_SECURITY_INFORMATION	The REG_SET_KEY_SECURITY_INFORMATION structure specifies security information for a registry key object.
REG_SET_VALUE_KEY_INFORMATION	The REG_SET_VALUE_INFORMATION structure describes a new setting for a registry key's value entry.
REG_UNLOAD_KEY_INFORMATION	The REG_UNLOAD_KEY_INFORMATION structure contains information that a driver's RegistryCallback routine can use when a registry hive is unloaded.
RESOURCEMANAGER_BASIC_INFORMATION	The RESOURCEMANAGER_BASIC INFORMATION structure contains information about a resource manager object.
RESOURCEMANAGER_COMPLETION_INFORMATION	The RESOURCEMANAGER_COMPLETION_INFORMATION structure is not used.
SCATTER_GATHER_LIST	The SCATTER_GATHER_LIST structure describes the scatter/gather list for a DMA operation.
SDEV_IDENTIFIER_INTERFACE	This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.
SECTION_OBJECT_POINTERS	The SECTION_OBJECT_POINTERS structure, allocated by a file system or a redirector driver, is used by the memory manager and cache manager to store file-mapping and cache-related information for a file stream.
SECURITY_SUBJECT_CONTEXT	The SECURITY_SUBJECT_CONTEXT structure is used to capture subject security context for access validation and auditing.
SLIST_ENTRY	An SLIST_ENTRY structure describes an entry in a sequenced singly linked list.
SYSTEM_POWER_STATE_CONTEXT	The SYSTEM_POWER_STATE_CONTEXT structure is a partially opaque system structure that contains information about the previous system power states of a computer.
TARGET_DEVICE_CUSTOM_NOTIFICATION	The TARGET_DEVICE_CUSTOM_NOTIFICATION structure describes a custom device event.
TARGET_DEVICE_REMOVAL_NOTIFICATION	The TARGET_DEVICE_REMOVAL_NOTIFICATION structure describes a device-removal event. The PnP manager sends this structure to a driver that registered a callback routine for notification of EventCategoryTargetDeviceChange events.
TRANSACTION_BASIC_INFORMATION	The TRANSACTION_BASIC_INFORMATION structure contains information about a transaction object.
TRANSACTION_ENLISTMENT_PAIR	The TRANSACTION_ENLISTMENT_PAIR structure contains information about an enlistment that is associated with a transaction object.

TITLE	DESCRIPTION
TRANSACTION_ENLISTMENTS_INFORMATION	The TRANSACTION_ENLISTMENTS_INFORMATION structure contains information about the enlistments that are associated with a transaction object.
TRANSACTION_PROPERTIES_INFORMATION	The TRANSACTION_PROPERTIES_INFORMATION structure contains a transaction object's properties.
TRANSACTIONMANAGER_BASIC_INFORMATION	The TRANSACTIONMANAGER_BASIC_INFORMATION structure contains information about a transaction manager object.
TRANSACTIONMANAGER_LOG_INFORMATION	The TRANSACTIONMANAGER_LOG_INFORMATION structure contains information about a transaction manager object.
TRANSACTIONMANAGER_LOGPATH_INFORMATION	The TRANSACTIONMANAGER_LOGPATH_INFORMATION structure contains information about a transaction manager object.
TRANSACTIONMANAGER_RECOVERY_INFORMATION	The TRANSACTIONMANAGER_RECOVERY_INFORMATION structure contains information about a transaction manager object.
VPB	The volume parameter block (VPB) structure is used to map a device object that represents a mounted file system volume to a device object that represents a physical or virtual disk device.
WAIT_CONTEXT_BLOCK	Reserved. Do not use.
WORK_QUEUE_ITEM	The WORK_QUEUE_ITEM structure is used to post a work items to a system work queue.
XSAVE_CET_U_FORMAT	Format for CET_U XSTATE component.

Enumerations

TITLE	DESCRIPTION
BOUND_CALLBACK_STATUS	The BOUND_CALLBACK_STATUS enumeration indicates how a user-mode bounds exception was processed by the BoundCallback function.
CLFS_CONTEXT_MODE	The CLFS_CONTEXT_MODE enumeration indicates the type of sequence that the Common Log File System (CLFS) driver follows when it reads a set of records from a stream.
CLFS_MGMT_POLICY_TYPE	The CLFS_MGMT_POLICY_TYPE enumeration type identifies the type of a CLFS management policy.
CLS_LOG_INFORMATION_CLASS	The CLFS_LOG_INFORMATION_CLASS enumeration indicates the type of information that is requested by a call to ClfsQueryLogFileInformation.

TITLE	DESCRIPTION
D3COLD_LAST_TRANSITION_STATUS	The D3COLD_LAST_TRANSITION_STATUS enumeration indicates whether the most recent transition to the D3hot device power state was followed by a transition to the D3cold device power state.
DEVICE_DIRECTORY_TYPE	The directory from which the driver is loaded.
DEVICE_INSTALL_STATE	The DEVICE_INSTALL_STATE enumeration describes a device's installation state.
DEVICE_POWER_STATE	The DEVICE_POWER_STATE enumeration type indicates a device power state.
DEVICE_REGISTRY_PROPERTY	The DEVICE_REGISTRY_PROPERTY enumeration identifies device properties that are stored in the registry.
DEVICE_REMOVAL_POLICY	The DEVICE_REMOVAL_POLICY enumeration describes a device's removal policy.
DEVICE_RESET_TYPE	The DEVICE_RESET_TYPE enumeration specifies the type of device reset that is being requested by a call to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface.
DEVICE_WAKE_DEPTH	The DEVICE_WAKE_DEPTH enumeration specifies the deepest device power state from which a device can trigger a wake signal.
DIRECTORY_NOTIFY_INFORMATION_CLASS	A value that specifies which structure to use to query or set information for a files in a directory.
DMA_COMPLETION_STATUS	The DMA_COMPLETION_STATUS enumeration describes the completion status of a DMA transfer.
DOMAIN_CONFIGURATION_ARCH	Defines values for type of system architecture specified.
DRIVER_DIRECTORY_TYPE	
ENLISTMENT_INFORMATION_CLASS	The ENLISTMENT_INFORMATION_CLASS enumeration identifies the type of information that the ZwSetInformationEnlistment routine can set and that the ZwQueryInformationEnlistment routine can retrieve for an enlistment object.
FAULT_INFORMATION_ARCH	Defines values for type of system architecture.
FAULT_INFORMATION_ARM64_TYPE	Defines values for the type of fault.
FILE_INFORMATION_CLASS	A value that specifies which structure to use to query or set information for a file object.
FS_INFORMATION_CLASS	The FS_INFORMATION_CLASS enumeration contains the information class constants that specify what type of information structure is present for a set or a query operation.

TITLE	DESCRIPTION
IMAGE_POLICY_ENTRY_TYPE	This enumeration is not supported.
IMAGE_POLICY_ID	This enumeration is not supported.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
IO_ACCESS_MODE	Defines the types of access mode for Scheduled File I/O (SFIO).
IO_ACCESS_TYPE	Defines the access rights for Scheduled File I/O (SFIO).
IO_ALLOCATION_ACTION	The IO_ALLOCATION_ACTION enumerated type is used to specify return values for AdapterControl and ControllerControl routines.
IO_CONTAINER_INFORMATION_CLASS	The IO_CONTAINER_INFORMATION_CLASS enumeration contains constants that indicate the classes of system information that a kernel-mode driver can request.
IO_CONTAINER_NOTIFICATION_CLASS	The IO_CONTAINER_NOTIFICATION_CLASS enumeration contains constants that indicate the classes of events for which a kernel-mode driver can register to receive notifications.
IO_PAGING_PRIORITY	The IO_PAGING_PRIORITY enumeration describes the priority value for a paging I/O IRP.
IO_PRIORITY_HINT	The IO_PRIORITY_HINT enumeration type specifies the priority hint for an IRP.
IO_SESSION_EVENT	The IO_SESSION_EVENT enumeration indicates the type of session event for which a driver is receiving notification.
IO_SESSION_STATE	The IO_SESSION_STATE enumeration contains constants that indicate the current state of a user session.
IRQ_DEVICE_POLICY	The IRQ_DEVICE_POLICY enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.
IRQ_PRIORITY	The IRQ_PRIORITY enumeration type indicates the priority the system should give to servicing a device's interrupts.
KBUGCHECK_CALLBACK_REASON	The KBUGCHECK_CALLBACK_REASON enumeration type specifies the situations in which a bug-check callback executes.
KBUGCHECK_DUMP_IO_TYPE	The KBUGCHECK_DUMP_IO_TYPE enumeration type identifies the type of a section of data within a crash dump file.
KEY_INFORMATION_CLASS	The KEY_INFORMATION_CLASS enumeration type represents the type of information to supply about a registry key.
KEY_SET_INFORMATION_CLASS	The KEY_SET_INFORMATION_CLASS enumeration type represents the type of information to set for a registry key.

TITLE	DESCRIPTION
KEY_VALUE_INFORMATION_CLASS	The KEY_VALUE_INFORMATION_CLASS enumeration type specifies the type of information to supply about the value of a registry key.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
KINTERRUPT_POLARITY	The KINTERRUPT_POLARITY enumeration indicates how a device signals an interrupt request on an interrupt line.
KTMOBJECT_TYPE	The KTMOBJECT_TYPE enumeration identifies the types of objects that KTM supports.
LOCK_OPERATION	The LOCK_OPERATION enumeration specifies the type of access that is appropriate for a type of I/O operation.
MEM_EXTENDED_PARAMETER_TYPE	Defines values for extended parameters are used for file mapping into an address space.
MEM_SECTION_EXTENDED_PARAMETER_TYPE	
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.
MONITOR_DISPLAY_STATE	Indicates the power state of the monitor being displayed on.
PCW_CALLBACK_TYPE	The PCW_CALLBACK_TYPE enumeration defines the notification type to send to the registered provider of the counter set. A provider passes a pointer to this enumeration as a parameter to the PcwCallback function.
PO_FX_PERF_STATE_TYPE	The PO_FX_PERF_STATE_TYPE enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.
PO_FX_PERF_STATE_UNIT	The PO_FX_PERF_STATE_UNIT enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.
POOL_TYPE	The POOL_TYPE enumeration type specifies the type of system memory to allocate.
POWER_ACTION	The POWER_ACTION enumeration identifies the system power actions that can occur on a computer.
POWER_INFORMATION_LEVEL	Indicates power level information.
POWER_REQUEST_TYPE	The POWER_REQUEST_TYPE enumeration indicates the power request type.
POWER_STATE_TYPE	The POWER_STATE_TYPE enumeration type indicates that a power state value is a system power state or a device power state.

TITLE	DESCRIPTION
REG_NOTIFY_CLASS	The REG_NOTIFY_CLASS enumeration type specifies the type of registry operation that the configuration manager is passing to a RegistryCallback routine.
RESOURCEMANAGER_INFORMATION_CLASS	The RESOURCEMANAGER_INFORMATION_CLASS enumeration identifies the type of information that the ZwQueryInformationResourceManager routine can retrieve for a resource manager object.
SE_IMAGE_TYPE	
SECURITY_IMPERSONATION_LEVEL	The SECURITY_IMPERSONATION_LEVEL enumeration type contains values that specify security impersonation levels. Security impersonation levels govern the degree to which a server process can act on behalf of a client process.
SYSTEM_POWER_STATE	The SYSTEM_POWER_STATE enumeration type is used to indicate a system power state.
TRACE_INFORMATION_CLASS	The TRACE_INFORMATION_CLASS enumeration type is used to indicate types of information associated with a WMI event tracing session.
TRANSACTION_INFORMATION_CLASS	The TRANSACTION_INFORMATION_CLASS enumeration specifies the type of information that ZwSetInformationTransaction can set and ZwQueryInformationTransaction can retrieve for a transaction manager object.
TRANSACTION_OUTCOME	The TRANSACTION_OUTCOME enumeration defines the outcomes (results) that KTM can assign to a transaction.
TRANSACTION_STATE	The TRANSACTION_STATE enumeration defines the states that KTM can assign to a transaction.
TRANSACTIONMANAGER_INFORMATION_CLASS	The TRANSACTIONMANAGER_INFORMATION_CLASS enumeration specifies the type of information that the ZwQueryInformationTransactionManager routine can retrieve for a transaction manager object.
WORK_QUEUE_TYPE	The WORK_QUEUE_TYPE enumeration type indicates the type of system worker thread that handles a work item.

DMA_COMPLETION_STATUS Enumeration

1/25/2019 • 2 minutes to read • Edit Online

The **DMA_COMPLETION_STATUS** enumeration describes the completion status of a DMA transfer.

Syntax

```
typedef enum  {
  DmaComplete,
  DmaAborted,
  DmaError,
  DmaCancelled
} DMA_COMPLETION_STATUS;
```

Constants

DmaComplete	The DMA transfer completed successfully.
DmaAborted	Not used.
DmaError	The DMA transfer did not complete successfully because an error occurred.
DmaCancelled	The DMA transfer did not complete successfully because the client canceled the transfer.

Remarks

The *Status* parameter of the DmaCompletionRoutine routine is a **DMA_COMPLETION_STATUS** enumeration value.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DmaCompletionRoutine

POWER_INFORMATION_LEVEL Enumeration

1/25/2019 • 3 minutes to read • Edit Online

Indicates power level information.

Syntax

```
typedef enum  {
  SystemPowerPolicyAc,
  SystemPowerPolicyDc,
  VerifySystemPolicyAc,
  VerifySystemPolicyDc,
  SystemPowerCapabilities,
  SystemBatteryState,
  SystemPowerStateHandler,
  ProcessorStateHandler,
  SystemPowerPolicyCurrent,
  AdministratorPowerPolicy,
  SystemReserveHiberFile,
  ProcessorInformation,
  SystemPowerInformation,
  ProcessorStateHandler2,
  LastWakeTime,
  LastSleepTime,
  SystemExecutionState,
  SystemPowerStateNotifyHandler,
  ProcessorPowerPolicyAc,
  ProcessorPowerPolicyDc,
  VerifyProcessorPowerPolicyAc,
  VerifyProcessorPowerPolicyDc,
  ProcessorPowerPolicyCurrent,
  SystemPowerStateLogging,
  SystemPowerLoggingEntry,
  SetPowerSettingValue,
  NotifyUserPowerSetting,
  PowerInformationLevelUnused0,
  SystemMonitorHiberBootPowerOff,
  SystemVideoState,
  TraceApplicationPowerMessage,
  TraceApplicationPowerMessageEnd,
  ProcessorPerfStates,
  ProcessorIdleStates,
  ProcessorCap,
  SystemWakeSource,
  SystemHiberFileInformation,
  TraceServicePowerMessage,
  ProcessorLoad,
  PowerShutdownNotification,
  MonitorCapabilities,
  SessionPowerInit,
  SessionDisplayState,
  PowerRequestCreate,
  PowerRequestAction,
  GetPowerRequestList,
  ProcessorInformationEx,
  NotifyUserModeLegacyPowerEvent,
  GroupPark,
  ProcessorIdleDomains,
  WakeTimerList,
  SystemHiberFileSize,
  ProcessorIdleStatesHv,
```

```
    ProcessorPerfStatesHv,
    ProcessorPerfCapHv,
    ProcessorSetIdle,
    LogicalProcessorIdling,
    UserPresence,
    PowerSettingNotificationName,
    GetPowerSettingValue,
    IdleResiliency,
    SessionRITState,
    SessionConnectNotification,
    SessionPowerCleanup,
    SessionLockState,
    SystemHiberbootState,
    PlatformInformation,
    PdcInvocation,
    MonitorInvocation,
    FirmwareTableInformationRegistered,
    SetShutdownSelectedTime,
    SuspendResumeInvocation,
    PlmPowerRequestCreate,
    ScreenOff,
    CsDeviceNotification,
    PlatformRole,
    LastResumePerformance,
    DisplayBurst,
    ExitLatencySamplingPercentage,
    RegisterSpmPowerSettings,
    PlatformIdleStates,
    ProcessorIdleVeto,
    PlatformIdleVeto,
    SystemBatteryStatePrecise,
    ThermalEvent,
    PowerRequestActionInternal,
    BatteryDeviceState,
    PowerInformationInternal,
    ThermalStandby,
    SystemHiberFileType,
    PhysicalPowerButtonPress,
    QueryPotentialDripsConstraint,
    EnergyTrackerCreate,
    EnergyTrackerQuery,
    UpdateBlackBoxRecorder,
    PowerInformationLevelMaximum
} POWER_INFORMATION_LEVEL;
```

Constants

SystemPowerPolicyAc	Indicates SystemPowerPolicyAc.
SystemPowerPolicyDc	Indicates SystemPowerPolicyDc.
VerifySystemPolicyAc	Indicates VerifySystemPolicyAc.
VerifySystemPolicyDc	Indicates VerifySystemPolicyDc.
SystemPowerCapabilities	Indicates the power capabilities of the system.
SystemBatteryState	Indicates the system's battery state.
SystemPowerStateHandler	Indicates the system's power state handler.

ProcessorStateHandler	Indicates the processor state handler.
SystemPowerPolicyCurrent	Indicates the system's current power policy.
AdministratorPowerPolicy	Indicates the administrator's power policy.
SystemReserveHiberFile	Indicates the SystemReserveHiberFile.
ProcessorInformation	Indicates the processor information.
SystemPowerInformation	Indicates the system power information.
ProcessorStateHandler2	Indicates the processor state handler.
LastWakeTime	Indicates the last wake time.
LastSleepTime	Indicates the last sleep time.
SystemExecutionState	Indicates the system execution state.
SystemPowerStateNotifyHandler	Indicates the system power state notify handler.
ProcessorPowerPolicyAc	Indicates ProcessorPowerPolicyAc.
ProcessorPowerPolicyDc	Indicates ProcessorPowerPolicyDc.
VerifyProcessorPowerPolicyAc	Indicates VerifyProcessorPowerPolicyAc.
VerifyProcessorPowerPolicyDc	Indicates VerifyProcessorPowerPolicyDc.
ProcessorPowerPolicyCurrent	Indicates the current processor power policy.
SystemPowerStateLogging	Indicates SystemPowerStateLogging.
SystemPowerLoggingEntry	Indicates SystemPowerLoggingEntry.
SetPowerSettingValue	Indicates that the power setting value is set.
NotifyUserPowerSetting	Indicates that the user should be notified of the power setting.
PowerInformationLevelUnused0	Indicates that the power information level is unused.
SystemMonitorHiberBootPowerOff	Indicates that the system monitor boot power is off.
SystemVideoState	Indicates the system video state.
TraceApplicationPowerMessage	Indicates the trace application power message.
TraceApplicationPowerMessageEnd	Indicates the end of the trace application power message.
ProcessorPerfStates	Indicates the processor performance states.

ProcessorIdleStates	Indicates the processor idle states.
ProcessorCap	Indicates the processor cap.
SystemWakeSource	Indicates the system wake source.
SystemHiberFileInformation	Indicates the system's hibernation file information.
TraceServicePowerMessage	Indicates the trace service power message.
ProcessorLoad	Indicates the processor load.
PowerShutdownNotification	Indicates the power shutdown notification.
MonitorCapabilities	Indicates the monitor's capabilities.
SessionPowerInit	Indicates the session power has been initialized.
SessionDisplayState	Indicates the session display state.
PowerRequestCreate	Indicates that a power request has been created.
PowerRequestAction	Indicates the action of the power request.
GetPowerRequestList	Indicates that the power request list should be queued.
ProcessorInformationEx	Indicates ProcessorInformationEx.
NotifyUserModeLegacyPowerEvent	Indicates that a notification should be created for the user mode legacy power event.
GroupPark	Indicates the group park.
ProcessorIdleDomains	Indicates the processor's idle domains.
WakeTimerList	Indicates the wake timer list.
SystemHiberFileSize	Indicates the system's hibernation file size.
ProcessorIdleStatesHv	Indicates the processor's idle states.
ProcessorPerfStatesHv	Indicates the processor's performance states.
ProcessorPerfCapHv	Indicates the processor's performance capabilities.
ProcessorSetIdle	Indicates that the processor has been set to idle.
LogicalProcessorIdling	Indicates that the processor is idling.
UserPresence	Indicates the user presence.
PowerSettingNotificationName	Indicates the power setting notification name.

GetPowerSettingValue	Indicates that the power setting value should be queued.
IdleResiliency	Indicates the idle resiliency.
SessionRITState	Indicates the session's RIT state.
SessionConnectNotification	Indicates the session's connect notification.
SessionPowerCleanup	Indicates the session's power cleanup.
SessionLockState	Indicates the session's lock state.
SystemHiberbootState	Indicates the system's hibernation boot state.
PlatformInformation	Indicates the platform information.
PdcInvocation	Indicates the pdc invocation.
MonitorInvocation	Indicates the monitor invocation.
FirmwareTableInformationRegistered	Indicates the registered firmware table information.
SetShutdownSelectedTime	Indicates that the shutdown time should be set.
SuspendResumeInvocation	Indicates SuspendResumeInvocation.
PlmPowerRequestCreate	Indicates that the power request has been created.
ScreenOff	Indicates that the screen is off.
CsDeviceNotification	Indicates the device notification.
PlatformRole	Indicates the platform role.
LastResumePerformance	Indicates the last time performance was resumed.
DisplayBurst	Indicates display burst.
ExitLatencySamplingPercentage	Indicates the latency sampling percentage.
RegisterSpmPowerSettings	Indicates that the power settings are registered.
PlatformIdleStates	Indicates the platform's idle states.
ProcessorIdleVeto	Indicates the processor's idle veto.
PlatformIdleVeto	Indicates the platform's idle veto.
SystemBatteryStatePrecise	Indicates the system's battery state.
ThermalEvent	Indicates the thermal event.

PowerRequestActionInternal	Indicates the internal power request action.
BatteryDeviceState	Indicates the battery's device state.
PowerInformationInternal	Indicates the internal power information.
ThermalStandby	Indicates thermal standby.
SystemHiberFileType	Indicates the system's hibernation file type.
PhysicalPowerButtonPress	Indicates a physical power button press.
QueryPotentialDripsConstraint	Indicates the potential drips constraint.
EnergyTrackerCreate	Indicates that the energy tracker is created.
EnergyTrackerQuery	Indicates that the energy tracker is queried.
UpdateBlackBoxRecorder	Indicates that the black box recorder is updated.
PowerInformationLevelMaximum	Indicates the maximum power level.

Requirements

Header	wdm.h (include Ntpoapi.h)

DEVICE_REGISTRY_PROPERTY Enumeration

1/25/2019 • 2 minutes to read • Edit Online

The **DEVICE_REGISTRY_PROPERTY** enumeration identifies device properties that are stored in the registry.

Syntax

```
typedef enum  {
  DevicePropertyDeviceDescription,
  DevicePropertyHardwareID,
  DevicePropertyCompatibleIDs,
  DevicePropertyBootConfiguration,
  DevicePropertyBootConfigurationTranslated,
  DevicePropertyClassName,
  DevicePropertyClassGuid,
  DevicePropertyDriverKeyName,
  DevicePropertyManufacturer,
  DevicePropertyFriendlyName,
  DevicePropertyLocationInformation,
  DevicePropertyPhysicalDeviceObjectName,
  DevicePropertyBusTypeGuid,
  DevicePropertyLegacyBusType,
  DevicePropertyBusNumber,
  DevicePropertyEnumeratorName,
  DevicePropertyAddress,
  DevicePropertyUINumber,
  DevicePropertyInstallState,
  DevicePropertyRemovalPolicy,
  DevicePropertyResourceRequirements,
  DevicePropertyAllocatedResources,
  DevicePropertyContainerID
} DEVICE_REGISTRY_PROPERTY;
```

Constants

DevicePropertyDeviceDescription	
DevicePropertyHardwareID	
DevicePropertyCompatibleIDs	
DevicePropertyBootConfiguration	
DevicePropertyBootConfigurationTranslated	
DevicePropertyClassName	
DevicePropertyClassGuid	
DevicePropertyDriverKeyName	
DevicePropertyManufacturer	

DevicePropertyFriendlyName	
DevicePropertyLocationInformation	
DevicePropertyPhysicalDeviceObjectName	
DevicePropertyBusTypeGuid	
DevicePropertyLegacyBusType	
DevicePropertyBusNumber	
DevicePropertyEnumeratorName	
DevicePropertyAddress	
DevicePropertyUINumber	
DevicePropertyInstallState	
DevicePropertyRemovalPolicy	
DevicePropertyResourceRequirements	
DevicePropertyAllocatedResources	
DevicePropertyContainerID	

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

IoGetDeviceProperty

BOOTDISK_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **BOOTDISK_INFORMATION** structure contains basic information describing the boot and system disks.

Syntax

```
typedef struct _BOOTDISK_INFORMATION {
  LONGLONG BootPartitionOffset;
  LONGLONG SystemPartitionOffset;
  ULONG    BootDeviceSignature;
  ULONG    SystemDeviceSignature;
} BOOTDISK_INFORMATION, *PBOOTDISK_INFORMATION;
```

Members

`BootPartitionOffset`

Specifies the offset, in bytes, on the boot disk where the boot partition begins.

`SystemPartitionOffset`

Specifies the offset, in bytes, on the system disk where the system partition begins.

`BootDeviceSignature`

If the boot disk is formatted with an MBR partition table, this specifies the signature for the disk's MBR partition table. Otherwise, this member is unused.

`SystemDeviceSignature`

If the system disk is formatted with an MBR partition table, this specifies the signature for the disk's MBR partition table. Otherwise, this member is unused.

Remarks

IoGetBootDiskInformation returns this structure to describe the boot and system disks.

Requirements

Minimum supported client	Drivers on Microsoft Windows XP and later versions of the Windows operating system should use the BOOTDISK_INFORMATION_EX structure instead.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BOOTDISK_INFORMATION_EX

IoGetBootDiskInformation

BOOTDISK_INFORMATION_EX structure

1/30/2019 • 2 minutes to read • Edit Online

The **BOOTDISK_INFORMATION_EX** structure contains extended information describing the boot and system disks.

Syntax

```
typedef struct _BOOTDISK_INFORMATION_EX {
  LONGLONG BootPartitionOffset;
  LONGLONG SystemPartitionOffset;
  ULONG    BootDeviceSignature;
  ULONG    SystemDeviceSignature;
  GUID     BootDeviceGuid;
  GUID     SystemDeviceGuid;
  BOOLEAN  BootDeviceIsGpt;
  BOOLEAN  SystemDeviceIsGpt;
} BOOTDISK_INFORMATION_EX, *PBOOTDISK_INFORMATION_EX;
```

Members

`BootPartitionOffset`

Specifies the offset, in bytes, on the boot disk where the boot partition begins.

`SystemPartitionOffset`

Specifies the offset, in bytes, on the system disk where the system partition begins.

`BootDeviceSignature`

If the **BootDeviceIsGpt** member is **FALSE**, this specifies the signature for the disk's MBR partition table. Otherwise, this member is unused.

`SystemDeviceSignature`

If the **SystemDeviceIsGpt** member is **FALSE**, this specifies the signature for the disk's MBR partition table. Otherwise, this member is unused.

`BootDeviceGuid`

If the **BootDeviceIsGpt** member is **TRUE**, this specifies the GUID for the boot disk. Otherwise, this member is unused.

`SystemDeviceGuid`

If the **SystemDeviceIsGpt** member is **TRUE**, this specifies the GUID for the boot disk. Otherwise, this member is unused.

`BootDeviceIsGpt`

TRUE if the boot disk is formatted with the GPT partition table type.

`SystemDeviceIsGpt`

TRUE if the system disk is formatted with the GPT partition table type.

Remarks

On Windows XP and later, IoGetBootDiskInformation returns this structure to describe the boot and system disks.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BOOTDISK_INFORMATION

IoGetBootDiskInformation

BOUND_CALLBACK_STATUS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **BOUND_CALLBACK_STATUS** enumeration indicates how a user-mode bounds exception was processed by the BoundCallback function.

Syntax

```
typedef enum _BOUND_CALLBACK_STATUS {
  BoundExceptionContinueSearch,
  BoundExceptionHandled,
  BoundExceptionError,
  BoundExceptionMaximum
} BOUND_CALLBACK_STATUS, *PBOUND_CALLBACK_STATUS;
```

Constants

BoundExceptionContinueSearch	The bounds exception was not handled by the callback, and the exception should continue to propagate.
BoundExceptionHandled	The exception was handled by the callback, and the exception should not propagate any further.
BoundExceptionError	The user mode process should be terminated by the system.
BoundExceptionMaximum	This value is not currently used.

Remarks

The return value of the BoundCallback routine is a **BOUND_CALLBACK_STATUS** value.

Requirements

Minimum supported client	Available starting with Windows 10.
Header	wdm.h

See Also

BoundCallback

BUS_INTERFACE_STANDARD structure

1/30/2019 • 2 minutes to read • Edit Online

The **BUS_INTERFACE_STANDARD** interface structure enables device drivers to make direct calls to parent bus driver routines. This structure defines the GUID_BUS_INTERFACE_STANDARD interface.

Syntax

```
typedef struct _BUS_INTERFACE_STANDARD {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
  PTRANSLATE_BUS_ADDRESS TranslateBusAddress;
  PGET_DMA_ADAPTER       GetDmaAdapter;
  PGET_SET_DEVICE_DATA   SetBusData;
  PGET_SET_DEVICE_DATA   GetBusData;
} BUS_INTERFACE_STANDARD, *PBUS_INTERFACE_STANDARD;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined interface version.

`Context`

A pointer to interface-specific context information.

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the interface's reference count.

`TranslateBusAddress`

A pointer to a TranslateBusAddress routine that translates addresses on the parent bus to logical addresses.

`GetDmaAdapter`

A pointer to a GetDmaAdapter routine that returns a DMA adapter structure (DMA_ADAPTER) for the target device.

`SetBusData`

A pointer to a SetBusData routine that writes data to the device's configuration space.

`GetBusData`

A pointer to a GetBusData routine that reads data from the device's configuration space.

Remarks

The **BUS_INTERFACE_STANDARD** structure is an extension of the INTERFACE structure.

Some operations on a device are reserved for the device's parent bus driver. These operations might include accessing the device-specific configuration space of a bus or programming a DMA controller.

To read from or write to a device's configuration space, a device driver must rely on the agency of the bus driver in either of two ways:

- By sending the I/O request packets (IRPs) IRP_MN_READ_CONFIG and IRP_MN_WRITE_CONFIG to the bus driver.
- By obtaining an interface from the bus driver. The device driver can then access its device's configuration space by making direct calls to the bus driver routines provided by the **BUS_INTERFACE_STANDARD** interface structure. Its member routines, GetBusData and SetBusData, can be used to read from and write to a device's configuration space, respectively.

For more information about the ways to access configuration space, see Accessing Device Configuration Space.

Some types of devices, such as a bus-mastering storage device, have on-board DMA controllers. However, the device drivers for these devices cannot program these DMA controllers directly. Instead they must rely on routines provided by the parent bus driver. For a device driver to program the DMA controller for its device, it must first request an adapter object from the parent bus driver. The adapter object contains the routines supplied by the bus driver that can be used to program the device's DMA controller. Device drivers must rely on the **BUS_INTERFACE_STANDARD**, either directly or indirectly, to obtain the adapter object.

If the driver is executing at IRQL = PASSIVE_LEVEL, it should obtain a device's DMA adapter object by calling IoGetDmaAdapter. **IoGetDmaAdapter** detects whether the bus driver supports the **BUS_INTERFACE_STANDARD** interface. If it does, **IoGetDmaAdapter** calls the routine pointed to by the GetDmaAdapter member of this interface to obtain the adapter object. Otherwise, **IoGetDmaAdapter** calls an equivalent legacy routine.

However, if a driver must obtain an adapter object while running at IRQL >= DISPATCH_LEVEL, it cannot do so with IoGetDmaAdapter. In this case, the driver must query for the **BUS_INTERFACE_STANDARD** interface while still at IRQL = PASSIVE_LEVEL by using IRP_MN_QUERY_INTERFACE.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DEVICE_DESCRIPTION

DMA_ADAPTER

GUID_BUS_INTERFACE_STANDARD

GetBusData

GetDmaAdapter

INTERFACE

IRP_MN_QUERY_INTERFACE

IRP_MN_READ_CONFIG

IRP_MN_WRITE_CONFIG

InterfaceDereference

InterfaceReference

IoGetDmaAdapter

SetBusData

TranslateBusAddress

CLFS_CONTEXT_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_CONTEXT_MODE** enumeration indicates the type of sequence that the Common Log File System (CLFS) driver follows when it reads a set of records from a stream.

Syntax

```
typedef enum _CLFS_CONTEXT_MODE {
  ClfsContextNone,
  ClfsContextUndoNext,
  ClfsContextPrevious,
  ClfsContextForward
} CLFS_CONTEXT_MODE, *PCLFS_CONTEXT_MODE, PPCLFS_CONTEXT_MODE;
```

Constants

ClfsContextNone	Indicates that a variable of type **CLFS_CONTEXT_MODE** has not yet been assigned a meaningful value.
ClfsContextUndoNext	Indicates that the next record in the sequence is pointed to by the undo-next LSN of the current record.
ClfsContextPrevious	Indicates that the next record in the sequence is pointed to by the previous LSN of the current record.
ClfsContextForward	Indicates that the next record in the sequence is the record in the stream that immediately follows the current record.

Requirements

Header	wdm.h (include Wdm.h)

See Also

ClfsReadLogRecord

ClfsReadNextLogRecord

ClfsReadPreviousRestartArea

ClfsReadRestartArea

CLFS_LOG_NAME_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_LOG_NAME_INFORMATION** structure holds the name of a Common Log File System (CLFS) stream or log.

Syntax

```
typedef struct _CLFS_LOG_NAME_INFORMATION {
  USHORT NameLengthInBytes;
  WCHAR  Name[1];
} CLFS_LOG_NAME_INFORMATION, *PCLFS_LOG_NAME_INFORMATION, PPCLFS_LOG_NAME_INFORMATION;
```

Members

`NameLengthInBytes`

The size, in bytes, of the log name.

`Name`

An array of wide characters that holds the log name.

Requirements

Header	wdm.h (include Wdm.h)

See Also

ClfsQueryLogFileInformation

ClfsSetLogFileInformation

CLFS_MGMT_CLIENT_REGISTRATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_MGMT_CLIENT_REGISTRATION** structure is given to CLFS management by clients who manage their own logs.

Syntax

```
typedef struct _CLFS_MGMT_CLIENT_REGISTRATION {
  ULONG                                    Version;
  PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK       AdvanceTailCallback;
  PVOID                                    AdvanceTailCallbackData;
  PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK LogGrowthCompleteCallback;
  PVOID                                    LogGrowthCompleteCallbackData;
  PCLFS_CLIENT_LOG_UNPINNED_CALLBACK       LogUnpinnedCallback;
  PVOID                                    LogUnpinnedCallbackData;
} CLFS_MGMT_CLIENT_REGISTRATION, *PCLFS_MGMT_CLIENT_REGISTRATION;
```

Members

`Version`

The version of the **CLFS_MGMT_CLIENT_REGISTRATION** structure. Set this to **CLFS_MGMT_CLIENT_REGISTRATION_VERSION**.

`AdvanceTailCallback`

A pointer to the log's ClfsAdvanceTailCallback function.

`AdvanceTailCallbackData`

A pointer to user-defined data that will be supplied to the *ClfsAdvanceTailCallback* function when the function is invoked.

`LogGrowthCompleteCallback`

A pointer to the log's ClfsLogGrowthCompleteCallback function.

`LogGrowthCompleteCallbackData`

A pointer to user-defined data that will be supplied to the *ClfsLogGrowthCompleteCallback* function when the function is invoked.

`LogUnpinnedCallback`

A pointer to the log's ClfsLogUnpinnedCallback function.

`LogUnpinnedCallbackData`

A pointer to user-defined data that will be supplied to the *ClfsLogUnpinnedCallback* function when the function is invoked.

Requirements

Header	wdm.h (include Wdm.h)

See Also

ClfsAdvanceTailCallback

ClfsLogGrowthCompleteCallback

ClfsLogUnpinnedCallback

CLFS_MGMT_POLICY structure

1/30/2019 • 3 minutes to read • Edit Online

The **CLFS_MGMT_POLICY** structure holds a description of a policy for managing a CLFS log.

Syntax

```
typedef struct _CLFS_MGMT_POLICY {
  ULONG                 Version;
  ULONG                 LengthInBytes;
  ULONG                 PolicyFlags;
  CLFS_MGMT_POLICY_TYPE PolicyType;
  union {
    struct {
      ULONG Containers;
    } MaximumSize;
    struct {
      ULONG Containers;
    } MinimumSize;
    struct {
      ULONG SizeInBytes;
    } NewContainerSize;
    struct {
      ULONG AbsoluteGrowthInContainers;
      ULONG RelativeGrowthPercentage;
    } GrowthRate;
    struct {
      ULONG MinimumAvailablePercentage;
      ULONG MinimumAvailableContainers;
    } LogTail;
    struct {
      ULONG Percentage;
    } AutoShrink;
    struct {
      ULONG Enabled;
    } AutoGrow;
    struct {
      USHORT PrefixLengthInBytes;
      WCHAR  PrefixString[1];
    } NewContainerPrefix;
    struct {
      ULONGLONG NextContainerSuffix;
    } NewContainerSuffix;
    struct {
      USHORT ExtensionLengthInBytes;
      WCHAR  ExtensionString[1];
    } NewContainerExtension;
  } PolicyParameters;
} CLFS_MGMT_POLICY, *PCLFS_MGMT_POLICY;
```

Members

`Version`

The version of the **CLFS_MGMT_POLICY** structure. Set this to **CLFS_MGMT_POLICY_VERSION**.

`LengthInBytes`

The length of the **CLFS_MGMT_POLICY** structure.

`PolicyFlags`

The flags that apply to this instance of the **CLFS_MGMT_POLICY** structure. The only flag that has been implemented for this release is **LOG_POLICY_OVERWRITE**, which indicates that when the policy is installed, it will replace the policy of the same type, if such a policy already exists.

`PolicyType`

A value of the CLFS_MGMT_POLICY_TYPE enumeration that supplies the type of this instance of the **CLFS_MGMT_POLICY** structure.

`PolicyParameters`

The union that provides the detailed information about this instance of the **CLFS_MGMT_POLICY** structure.

`PolicyParameters.MaximumSize`

`PolicyParameters.MaximumSize.Containers`

The maximum number of containers that the log will use.

`PolicyParameters.MinimumSize`

`PolicyParameters.MinimumSize.Containers`

The minimum number of containers that the log will use.

`PolicyParameters.NewContainerSize`

`PolicyParameters.NewContainerSize.SizeInBytes`

The size of each of the log's containers.

`PolicyParameters.GrowthRate`

`PolicyParameters.GrowthRate.AbsoluteGrowthInContainers`

The number of containers that should be added when the size of the log is increased. If the **RelativeGrowthPercentage** member is nonzero, then **AbsoluteGrowthInContainers** must be zero.

`PolicyParameters.GrowthRate.RelativeGrowthPercentage`

The percentage by which the log's size should increase when the log grows, expressed as a number between zero and 100. For example, if the log consisted of 32 containers and **RelativeGrowthPercentage** was ten, then, when the log needed to grow, it would grow by three (32 * 10 percent, rounded down to the nearest integer) containers. If the **AbsoluteGrowthInContainers** member is nonzero, then **RelativeGrowthPercentage** must be zero.

`PolicyParameters.LogTail`

`PolicyParameters.LogTail.MinimumAvailablePercentage`

When CLFS management notifies the client to move its log tail, it will specify that the tail be moved to an LSN that leaves at least **MinimumAvailablePercentage** percent of the log free. If the **MinimumAvailableContainers** member is nonzero, then **MinimumAvailablePercentage** must be zero.

`PolicyParameters.LogTail.MinimumAvailableContainers`

When CLFS management notifies the client to move its log tail, it will specify that the tail be moved to an LSN that leaves at least **MinimumAvailableContainers** containers free. If the **MinimumAvailablePercentage** member is nonzero, then **MinimumAvailableContainers** must be zero.

`PolicyParameters.AutoShrink`

`PolicyParameters.AutoShrink.Percentage`

When the percentage of free space in the log reaches **Percentage**, the log will shrink. Percentage is expressed as a number between 0 and 100, so a value of 25 would mean 25 percent.

`PolicyParameters.AutoGrow`

`PolicyParameters.AutoGrow.Enabled`

A numeric value that determines whether automatic log growth is enabled. Any nonzero value enables automatic growth.

`PolicyParameters.NewContainerPrefix`

`PolicyParameters.NewContainerPrefix.PrefixLengthInBytes`

The length, in bytes, of the **PrefixString** member.

`PolicyParameters.NewContainerPrefix.PrefixString`

A wide-character string that contains the full path to the directory where the log's containers reside, as well as a prefix that will be used as part of the file name for each container in the log.

`PolicyParameters.NewContainerSuffix`

`PolicyParameters.NewContainerSuffix.NextContainerSuffix`

The number to use as the suffix of the file name for the next container in the log. To form the file name, the number is converted to a string of decimal digits and appended to the prefix string. The number is incremented for the file name of each subsequent container.

`PolicyParameters.NewContainerExtension`

`PolicyParameters.NewContainerExtension.ExtensionLengthInBytes`

The length, in bytes, of the **ExtensionString** member.

`PolicyParameters.NewContainerExtension.ExtensionString`

A wide-character string that contains the extension to the file name for each container in the log. Container file names are built using the format [prefix][suffix][.extension]. An extension is optional. The default extension is the empty string.

Remarks

The way a **CLFS_MGMT_POLICY** structure is interpreted depends on the type of policy that the structure holds.

You can provide *policies* that specify how the log will be managed. Each policy is an instance of the **CLFS_MGMT_POLICY** structure, but the structure is interpreted differently depending on the policy type. CLFS uses the information that you provided in the policies to tailor how it manages the log.

When you create a **CLFS_MGMT_POLICY** structure whose **PolicyType** is **ClfsMgmtPolicyNewContainerPrefix**, be sure to allocate enough space to hold the **PolicyParameters.NewContainerPrefix.PrefixString** string.

You can only install a policy whose policy type specified in the **PolicyType** value is **ClfsMgmtPolicyNewContainerSize** before there are any containers in the log. You can change other policies after the log exists.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CLFS_MGMT_POLICY_TYPE

ClfsMgmtInstallPolicy

ClfsMgmtQueryPolicy

ClfsMgmtRemovePolicy

CLFS_MGMT_POLICY_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_MGMT_POLICY_TYPE** enumeration type identifies the type of a CLFS management policy.

Syntax

```
typedef enum _CLFS_MGMT_POLICY_TYPE {
  ClfsMgmtPolicyMaximumSize,
  ClfsMgmtPolicyMinimumSize,
  ClfsMgmtPolicyNewContainerSize,
  ClfsMgmtPolicyGrowthRate,
  ClfsMgmtPolicyLogTail,
  ClfsMgmtPolicyAutoShrink,
  ClfsMgmtPolicyAutoGrow,
  ClfsMgmtPolicyNewContainerPrefix,
  ClfsMgmtPolicyNewContainerSuffix,
  ClfsMgmtPolicyNewContainerExtension,
  ClfsMgmtPolicyInvalid
} CLFS_MGMT_POLICY_TYPE, *PCLFS_MGMT_POLICY_TYPE;
```

Constants

ClfsMgmtPolicyMaximumSize	Indicates a policy that specifies the maximum size of a log.
ClfsMgmtPolicyMinimumSize	Indicates a policy that specifies the minimum size of a log.
ClfsMgmtPolicyNewContainerSize	Indicates a policy that specifies the size of new containers that are created.
ClfsMgmtPolicyGrowthRate	Indicates a policy that specifies how many new containers will be added to the log each time the log grows.
ClfsMgmtPolicyLogTail	Indicates a policy that specifies how much free space will be requested when a client is notified to move its log tail.
ClfsMgmtPolicyAutoShrink	Indicates a policy that specifies when the log will shrink based on the percentage of the log that is free.
ClfsMgmtPolicyAutoGrow	Indicates a policy that specifies whether the log should grow when fewer than two containers are free.
ClfsMgmtPolicyNewContainerPrefix	Indicates a policy that specifies a prefix for the file name of each container, as well as the full path to the directory where the containers will be placed.
ClfsMgmtPolicyNewContainerSuffix	Indicates a policy that specifies a number to use as the starting suffix for container file names.
ClfsMgmtPolicyNewContainerExtension	Indicates a policy that specifies an extension for the file name of each container.

ClfsMgmtPolicyInvalid	Reserved for internal use.

Remarks

Each type of CLFS management policy corresponds to a specific interpretation of the CLFS_MGMT_POLICY structure. The **PolicyType** member of the **CLFS_MGMT_POLICY** structure is a valid value of the **CLFS_MGMT_POLICY_TYPE** enumeration.

Requirements

Header	wdm.h (include Wdm.h)

See Also

CLFS_MGMT_POLICY

CLFS_STREAM_ID_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_STREAM_ID_INFORMATION** structure holds a value that identifies a stream in a Common Log File System (CLFS) log.

Syntax

```
typedef struct _CLFS_STREAM_ID_INFORMATION {
  UCHAR StreamIdentifier;
} CLFS_STREAM_ID_INFORMATION, *PCLFS_STREAM_ID_INFORMATION, PPCLFS_STREAM_ID_INFORMATION;
```

Members

`StreamIdentifier`

An 8-bit value that identifies a stream.

Remarks

A stream identifier is unique within a given CLFS log.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ClfsQueryLogFileInformation

CLS_CONTAINER_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_CONTAINER_INFORMATION** structure holds descriptive information for an individual container in a Common Log File System (CLFS) log.

Syntax

```
typedef struct _CLS_CONTAINER_INFORMATION {
  ULONG                FileAttributes;
  ULONGLONG            CreationTime;
  ULONGLONG            LastAccessTime;
  ULONGLONG            LastWriteTime;
  LONGLONG             ContainerSize;
  ULONG                FileNameActualLength;
  ULONG                FileNameLength;
  WCHAR                FileName[CLFS_MAX_CONTAINER_INFO];
  CLFS_CONTAINER_STATE State;
  CLFS_CONTAINER_ID    PhysicalContainerId;
  CLFS_CONTAINER_ID    LogicalContainerId;
} CLS_CONTAINER_INFORMATION, *PCLS_CONTAINER_INFORMATION, PPCLS_CONTAINER_INFORMATION;
```

Members

`FileAttributes`

A set of flags that specifies attributes of the container. See the *fFlagsAndAttributes* parameter of the ClfsCreateLogFile function.

`CreationTime`

The time that the container was created.

`LastAccessTime`

The time that the container was last accessed.

`LastWriteTime`

The time of the last write to the container.

`ContainerSize`

The size, in bytes, of the container.

`FileNameActualLength`

The size, in characters, of the actual file name of the container.

`FileNameLength`

The size of the file name in the **FileName** buffer.

`FileName`

An array of wide characters that holds the file name of the container.

`State`

An integer that specifies the state of the container. This member must be one of the following values.

VALUE	MEANING
ClfsContainerInitializing	The container has not yet been initialized.
ClfsContainerInactive	The container has been initialized but does not hold any records that are in the active portion of the log.
ClfsContainerActive	The container is being used to hold records that belong to the active portion of the log.
ClfsContainerActivePendingDelete	The container is marked for deletion, but still holds records that belong to the active portion of the log.
ClfsContainerPendingArchive	The container is pending archival.
ClfsContainerPendingArchiveAndDelete	The container is marked for deletion, but still contains records that are pending archival.

`PhysicalContainerId`

A 32-bit identifier that remains the same over the life of the log.

`LogicalContainerId`

A 32-bit identifier that changes every time the container is recycled.

Remarks

The ClfsScanLogContainers function writes descriptive information into an array of **CLFS_CONTAINER_INFORMATION** structures.

Time values **CreationTime**, **LastAccessTime**, and **LastWriteTime** are expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CLFS_SCAN_CONTEXT

ClfsCreateLogFile

ClfsCreateScanContext

ClfsScanLogContainers

CLS_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_INFORMATION** structure holds metadata and state information for a Common Log File System (CLFS) stream and/or its underlying physical log.

Syntax

```
typedef struct _CLS_INFORMATION {
  LONGLONG  TotalAvailable;
  LONGLONG  CurrentAvailable;
  LONGLONG  TotalReservation;
  ULONGLONG BaseFileSize;
  ULONGLONG ContainerSize;
  ULONG     TotalContainers;
  ULONG     FreeContainers;
  ULONG     TotalClients;
  ULONG     Attributes;
  ULONG     FlushThreshold;
  ULONG     SectorSize;
  CLS_LSN   MinArchiveTailLsn;
  CLS_LSN   BaseLsn;
  CLS_LSN   LastFlushedLsn;
  CLS_LSN   LastLsn;
  CLS_LSN   RestartLsn;
  GUID      Identity;
} CLS_INFORMATION, *PCLS_INFORMATION, *PPCLS_INFORMATION;
```

Members

`TotalAvailable`

The total available space allocated to the log. This is calculated as the sum of the sizes of all containers in the log.

`CurrentAvailable`

The amount of space available in the log for new records and reservation allocations. This space is the total available space minus the undo commitment space and space used for storing owner pages in a multiplexed log.

`TotalReservation`

The amount of space reserved in the stream (or physical log) for undo operations.

`BaseFileSize`

The size, in bytes, of the base log file.

`ContainerSize`

The size, in bytes, of an individual container in the log. Note that all containers in the log are the same size.

`TotalContainers`

The number of containers in the log.

`FreeContainers`

The number of containers in the log that are not active.

`TotalClients`

The number of streams that share the log.

`Attributes`

A set of flags that specify stream (or physical log) attributes. See the *fFlagsAndAttributes* parameter of the ClfsCreateLogFile function.

`FlushThreshold`

The number of bytes of data (including headers) that are allowed to remain pending on the internal flush queue before CLFS automatically schedules a thread to write the flush queue to stable storage.

`SectorSize`

The sector size, in bytes, of the underlying disk geometry. The sector size is assumed to be a multiple of 512 and consistent across containers.

`MinArchiveTailLsn`

The LSN of the oldest record in the log for which archiving has not taken place. The minimum of this and the base LSN determines the last container that can be reused when containers are recycled.

`BaseLsn`

The LSN of the oldest record in the stream (or physical log) that is still needed by the stream (or log) clients.

`LastFlushedLsn`

The LSN of the last record that was flushed to stable storage.

`LastLsn`

The LSN of the youngest record in the stream (or physical log) that is still needed by the stream (or log) clients.

`RestartLsn`

The LSN of the last restart record written to the stream (or physical log). If there are no restart records, this member is equal to CLFS_LSN_INVALID.

`Identity`

A GUID that serves as a unique identifier for the log.

Remarks

The ClfsQueryLogFileInformation function returns information about a CLFS stream and/or its underlying physical log. The type of information returned depends on the *eInformationClass* parameter.

If the *eInformationClass* parameter is equal to **ClfsLogBasicInformationPhysical**, **ClfsQueryLogFileInformation** returns information in a **CLFS_INFORMATION** structure, and every structure member holds a piece of information about a physical CLFS log. Even if the log is multiplexed (that is, it has several streams), all structure members hold information about the underlying physical log rather than information about one of the streams.

If *eInformationClass* parameter is equal to **ClfsLogBasicInformation**, **ClfsQueryLogFileInformation** returns information in a **CLFS_INFORMATION** structure. Most of the structure members hold information about a CLFS physical log, but if the log is multiplexed, certain members hold information that is specific to a particular stream.

The following structure members hold information that is specific to a stream rather than the underlying physical log:

- **TotalUndoCommitment**
- **Attributes**
- **BaseLsn**
- **LastLsn**
- **RestartLsn**

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ClfsCreateLogFile

ClfsQueryLogFileInformation

ClfsSetLogFileInformation

CLS_IO_STATISTICS structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_IO_STATISTICS** structure holds I/O statistics data for a Common Log File System (CLFS) log.

Syntax

```
typedef struct _CLS_IO_STATISTICS {
  CLS_IO_STATISTICS_HEADER hdrIoStats;
  ULONGLONG                cFlush;
  ULONGLONG                cbFlush;
  ULONGLONG                cMetaFlush;
  ULONGLONG                cbMetaFlush;
} CLS_IO_STATISTICS, *PCLS_IO_STATISTICS, PPCLS_IO_STATISTICS;
```

Members

`hdrIoStats`

A CLFS_IO_STATISTICS_HEADER structure that holds header information for the set of statistics.

`cFlush`

The number of data flushes.

`cbFlush`

The number of bytes of data flushed.

`cMetaFlush`

The number of metadata flushes.

`cbMetaFlush`

The number of bytes of metadata flushed.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntddk.h)

See Also

CLFS_IO_STATISTICS_HEADER

ClfsGetIoStatistics

CLS_IO_STATISTICS_HEADER structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_IO_STATISTICS_HEADER** structure holds the header portion of a CLFS_IO_STATISTICS structure.

Syntax

```
typedef struct _CLS_IO_STATISTICS_HEADER {
  UCHAR              ubMajorVersion;
  UCHAR              ubMinorVersion;
  CLFS_IOSTATS_CLASS eStatsClass;
  USHORT             cbLength;
  ULONG              coffData;
} CLS_IO_STATISTICS_HEADER, *PCLS_IO_STATISTICS_HEADER, PPCLS_IO_STATISTICS_HEADER;
```

Members

`ubMajorVersion`

The major version of the CLFS_IO_STATISTICS structure.

`ubMinorVersion`

The minor version of the **CLFS_IO_STATISTICS** structure.

`eStatsClass`

Reserved for future use. This member is ignored.

`cbLength`

The size, in bytes, of the **CLFS_IO_STATISTICS** structure, including the header.

`coffData`

The offset, in bytes, from the beginning of the **CLFS_IO_STATISTICS** structure to the beginning of the statistics data. This member allows for transparent modifications to the header.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CLFS_IO_STATISTICS

ClfsGetIoStatistics

CLS_LOG_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_LOG_INFORMATION_CLASS** enumeration indicates the type of information that is requested by a call to ClfsQueryLogFileInformation.

Syntax

```
typedef enum _CLS_LOG_INFORMATION_CLASS {
  ClfsLogBasicInformation,
  ClfsLogBasicInformationPhysical,
  ClfsLogPhysicalNameInformation,
  ClfsLogStreamIdentifierInformation,
  ClfsLogSystemMarkingInformation,
  ClfsLogPhysicalLsnInformation
} CLS_LOG_INFORMATION_CLASS, *PCLS_LOG_INFORMATION_CLASS, PPCLS_LOG_INFORMATION_CLASS;
```

Constants

ClfsLogBasicInformation	Indicates that the request is for basic information about a CLFS stream and its underlying physical log. The information is returned in a CLFS_INFORMATION structure. Most of the structure members contain information about the underlying physical log, but some members contain information that is specific to the stream.
ClfsLogBasicInformationPhysical	Indicates that the request is for basic information about the physical log that underlies a CLFS stream. The information is returned in a **CLFS_INFORMATION** structure. All of the structure members contain information about the underlying physical log.
ClfsLogPhysicalNameInformation	Indicates that the request is for information about the name of a physical CLFS log. The information is returned in a **CLFS_LOG_NAME_INFORMATION** structure.
ClfsLogStreamIdentifierInformation	Indicates that the request is for a CLFS stream identifier. The information is returned in a CLFS_STREAM_ID_INFORMATION structure.
ClfsLogSystemMarkingInformation	Count of system marking references. This enumeration constant is supported only in Windows Vista and later versions of Windows.
ClfsLogPhysicalLsnInformation	Maps virtual LSNs to physical LSNs; only valid for physical logs. This enumeration constant is supported only in Windows Vista and later versions of Windows.

Remarks

The *eInformationClass* parameter of the ClfsQueryLogFileInformation function is a value from the **CLFS_LOG_INFORMATION_CLASS** enumeration.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ClfsQueryLogFileInformation

CLS_LSN structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_LSN** structure identifies an individual record in a Common Log File System (CLFS) stream.

Syntax

```
typedef struct _CLS_LSN {
  ULONGLONG Internal;
} CLS_LSN, *PCLS_LSN, PPCLS_LSN;
```

Members

`Internal`

A 64-bit value that holds three pieces of information about a log record: container identifier, block offset, and record sequence number.

Remarks

A container is a contiguous physical disk extent that serves as part of a CLFS log. A given CLFS log has several containers. Each container has several sector-aligned blocks, each of which holds a numbered sequence of records. A record can be identified by the triple (logical container identifier, block offset, record sequence number).

To obtain the container identifier, block offset, and record sequence number from a **CLFS_LSN** structure, call ClfsLsnContainer, ClfsLsnBlockOffset, and ClfsLsnRecordSequence.

Requirements

Header	wdm.h (include Wdm.h)

CLS_SCAN_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_SCAN_CONTEXT** structure holds context information to support a scan of the containers in a Common Log File System (CLFS) log. The client allocates this structure by calling ClfsCreateScanContext. Then, the client passes the structure repeatedly to ClfsScanLogContainers.

Syntax

```
typedef struct _CLS_SCAN_CONTEXT {
  CLFS_NODE_ID               cidNode;
  PLOG_FILE_OBJECT           plfoLog;
  ULONG                      cIndex;
  ULONG                      cContainers;
  ULONG                      cContainersReturned;
  CLFS_SCAN_MODE             eScanMode;
  PCLS_CONTAINER_INFORMATION pinfoContainer;
} CLS_SCAN_CONTEXT, *PCLS_SCAN_CONTEXT, PPCLS_SCAN_CONTEXT;
```

Members

`cidNode`

A CLFS_NODE_ID structure that contains the scan context's node identity and type.

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS log.

`cIndex`

The index of the current container.

`cContainers`

The number of containers to scan in each call to **ClfsScanLogContainers**. This is the number of elements in the array pointed to by *pinfoContainer*.

`cContainersReturned`

The number of containers actually scanned in a call to **ClfsScanLogContainers**.

`eScanMode`

This member can be one of the following values.

VALUE	MEANING
CLFS_SCAN_INIT	The scan is reinitialized and begins at the first container in the container list.
CLFS_SCAN_FORWARD	The next set of containers is scanned.
CLFS_SCAN_BACKWARD	The previous set of containers is scanned.

CLFS_SCAN_CLOSE	The scan is closed. No containers are scanned.

`pinfoContainer`

A pointer to an array of CLFS_CONTAINER_INFORMATION structures. The *cContainers* parameter specifies the number of elements in this array.

Requirements

Header	wdm.h (include Wdm.h)

CLS_WRITE_ENTRY structure

1/30/2019 • 2 minutes to read • Edit Online

The **CLFS_WRITE_ENTRY** structure holds the address and size of a buffer that contains one unit of data to be written to a Common Log File System (CLFS) stream.

Syntax

```
typedef struct _CLS_WRITE_ENTRY {
  PVOID Buffer;
  ULONG ByteLength;
} CLS_WRITE_ENTRY, *PCLS_WRITE_ENTRY, PPCLS_WRITE_ENTRY;
```

Members

`Buffer`

A pointer to the buffer that holds the data.

`ByteLength`

The size, in bytes, of the buffer pointed to by **Buffer**.

Remarks

CLFS collects the buffers pointed to by several **CLFS_WRITE_ENTRY** structures into a single log record. The log record is then placed in a log I/O block along with other log records. Eventually the log I/O block is flushed to disk.

Requirements

Header	wdm.h (include Wdm.h)

See Also

ClfsReserveAndAppendLog

ClfsReserveAndAppendLogAligned

CM_EISA_FUNCTION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_EISA_FUNCTION_INFORMATION** structure defines detailed EISA configuration information returned by HalGetBusData for the input *BusDataType***EisaConfiguration**, or by **HalGetBusDataByOffset** for the input *BusDataType***EisaConfiguration** and the *Offset* zero, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _CM_EISA_FUNCTION_INFORMATION {
  ULONG                     CompressedId;
  UCHAR                     IdSlotFlags1;
  UCHAR                     IdSlotFlags2;
  UCHAR                     MinorRevision;
  UCHAR                     MajorRevision;
  UCHAR                     Selections[26];
  UCHAR                     FunctionFlags;
  UCHAR                     TypeString[80];
  EISA_MEMORY_CONFIGURATION EisaMemory[9];
  EISA_IRQ_CONFIGURATION    EisaIrq[7];
  EISA_DMA_CONFIGURATION    EisaDma[4];
  EISA_PORT_CONFIGURATION   EisaPort[20];
  UCHAR                     InitializationData[60];
} CM_EISA_FUNCTION_INFORMATION, *PCM_EISA_FUNCTION_INFORMATION;
```

Members

`CompressedId`

The EISA compressed identification of the device at this slot. The value is identical to the **CompressedId** member of the CM_EISA_SLOT_INFORMATION structure.

`IdSlotFlags1`

The EISA slot identification flags.

`IdSlotFlags2`

The EISA slot identification flags.

`MinorRevision`

Information supplied by the manufacturer.

`MajorRevision`

Information supplied by the manufacturer.

`Selections`

The EISA selections for the device.

`FunctionFlags`

Indicates which of the members has available information. Callers can use the following system-defined masks to determine whether a particular type of configuration information can be or has been returned by **HalGetBusData**

or HalGetBusDataByOffset:

EISA_FUNCTION_ENABLED

EISA_FREE_FORM_DATA

EISA_HAS_PORT_INIT_ENTRY

EISA_HAS_PORT_RANGE

EISA_HAS_DMA_ENTRY

EISA_HAS_IRQ_ENTRY

EISA_HAS_MEMORY_ENTRY

EISA_HAS_TYPE_ENTRY

EISA_HAS_INFORMATION

The EISA_HAS_INFORMATION mask is a combination of the following:

EISA_HAS_PORT_RANGE

EISA_HAS_DMA_ENTRY

EISA_HAS_IRQ_ENTRY

EISA_HAS_MEMORY_ENTRY

EISA_HAS_TYPE_ENTRY

`TypeString`

Specifies the type of device.

`EisaMemory`

Describes the EISA device memory configuration information, defined as follows:

```
typedef struct _EISA_MEMORY_CONFIGURATION {
    EISA_MEMORY_TYPE ConfigurationByte;
    UCHAR DataSize;
    USHORT AddressLowWord;
    UCHAR AddressHighByte;
    USHORT MemorySize;
} EISA_MEMORY_CONFIGURATION, *PEISA_MEMORY_CONFIGURATION;
```

`EisaIrq`

Describes the EISA interrupt configuration information, defined as follows:

```
typedef struct _EISA_IRQ_CONFIGURATION {
    EISA_IRQ_DESCRIPTOR ConfigurationByte;
    UCHAR Reserved;
} EISA_IRQ_CONFIGURATION, *PEISA_IRQ_CONFIGURATION;
```

`EisaDma`

Describes the EISA DMA configuration information, defined as follows:

```
typedef struct _EISA_DMA_CONFIGURATION {
    DMA_CONFIGURATION_BYTE0 ConfigurationByte0;
    DMA_CONFIGURATION_BYTE1 ConfigurationByte1;
} EISA_DMA_CONFIGURATION, *PEISA_DMA_CONFIGURATION;
```

`EisaPort`

Describes the EISA device port configuration information, defined as follows:

```
typedef struct _EISA_PORT_CONFIGURATION {
    EISA_PORT_DESCRIPTOR Configuration;
    USHORT PortAddress;
} EISA_PORT_CONFIGURATION, *PEISA_PORT_CONFIGURATION;
```

`InitializationData`

Vendor-supplied, device-specific initialization data, if any.

Remarks

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in **CM_EISA_FUNCTION_INFORMATION** and/or in the CM_EISA_SLOT_INFORMATION header immediately preceding it is read-only.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_EISA_SLOT_INFORMATION

HalGetBusData

HalGetBusDataByOffset

CM_EISA_SLOT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_EISA_SLOT_INFORMATION** structure defines EISA configuration header information returned by HalGetBusData for the input *BusDataType* = **EisaConfiguration**, or by HalGetBusDataByOffset for the inputs *BusDataType* = **EisaConfiguration** and *Offset* = 0, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _CM_EISA_SLOT_INFORMATION {
  UCHAR  ReturnCode;
  UCHAR  ReturnFlags;
  UCHAR  MajorRevision;
  UCHAR  MinorRevision;
  USHORT Checksum;
  UCHAR  NumberFunctions;
  UCHAR  FunctionInformation;
  ULONG  CompressedId;
} CM_EISA_SLOT_INFORMATION, *PCM_EISA_SLOT_INFORMATION;
```

Members

`ReturnCode`

Contains a status code if an error occurs when the EISA BIOS is queried. Possible status codes include the following:

- EISA_INVALID_SLOT
- EISA_INVALID_FUNCTION
- EISA_INVALID_CONFIGURATION
- EISA_EMPTY_SLOT
- EISA_INVALID_BIOS_CALL

`ReturnFlags`

The return flags.

`MajorRevision`

Information supplied by the manufacturer.

`MinorRevision`

Information supplied by the manufacturer.

`Checksum`

The checksum value, allowing validation of the configuration data.

`NumberFunctions`

The number at this slot.

`FunctionInformation`

Whether there is available CM_EISA_FUNCTION_INFORMATION for this slot.

`CompressedId`

The EISA compressed identification of the device at this slot. This value is identical to the **CompressedId** member of the **CM_EISA_FUNCTION_INFORMATION** structure. This member can be read to determine whether the caller should call **HalGetBusData** or **HalGetBusDataByOffset** again with sufficient buffer space to get more detailed **CM_EISA_FUNCTION_INFORMATION** for a device it supports.

Remarks

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in **CM_EISA_SLOT_INFORMATION** and in **CM_EISA_FUNCTION_INFORMATION** immediately following it is read-only.

The driver of an EISA device might call **HalGetBusData** or **HalGetBusDataByOffset** for each slot on each EISA bus in the system, requesting only **CM_EISA_SLOT_INFORMATION** in order to find the device(s) it supports by examining the returned **CompressedId** values. Then, such a driver could allocate sufficient buffer space to call **HalGetBusData(ByOffset)** again for **CM_EISA_SLOT_INFORMATION** and **CM_EISA_FUNCTION_INFORMATION** at slots where its device(s) can be found.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_EISA_FUNCTION_INFORMATION

HalGetBusData

HalGetBusDataByOffset

CM_FLOPPY_DEVICE_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_FLOPPY_DEVICE_DATA** structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a floppy controller if the system can collect this information during the boot process.

Syntax

```
typedef struct _CM_FLOPPY_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  CHAR   Size[8];
  ULONG  MaxDensity;
  ULONG  MountDensity;
  UCHAR  StepRateHeadUnloadTime;
  UCHAR  HeadLoadTime;
  UCHAR  MotorOffTime;
  UCHAR  SectorLengthCode;
  UCHAR  SectorPerTrack;
  UCHAR  ReadWriteGapLength;
  UCHAR  DataTransferLength;
  UCHAR  FormatGapLength;
  UCHAR  FormatFillCharacter;
  UCHAR  HeadSettleTime;
  UCHAR  MotorSettleTime;
  UCHAR  MaximumTrackValue;
  UCHAR  DataTransferRate;
} CM_FLOPPY_DEVICE_DATA, *PCM_FLOPPY_DEVICE_DATA;
```

Members

`Version`

The version number of this structure.

`Revision`

The revision of this structure.

`Size`

The floppy disk density size.

`MaxDensity`

The maximum density.

`MountDensity`

The mount density.

`StepRateHeadUnloadTime`

The step rate head unload time in milliseconds.

`HeadLoadTime`

The head load time in milliseconds.

`MotorOffTime`

The motor off time in seconds.

`SectorLengthCode`

Indicates the sector size as an exponent in the formula ((2**_code_) * 128).

`SectorPerTrack`

The number of sectors per track.

`ReadWriteGapLength`

The read/write gap length, in bytes.

`DataTransferLength`

The data transfer length, in bytes, not including the synchronization field.

`FormatGapLength`

The format gap length, in bytes.

`FormatFillCharacter`

The format fill character.

`HeadSettleTime`

The head settle time in milliseconds.

`MotorSettleTime`

The motor settle time in milliseconds.

`MaximumTrackValue`

The maximum track number on the media. Track numbers are zero-based values.

`DataTransferRate`

The value written to the Datarate register before accessing the media.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IoQueryDeviceDescription

IoReportResourceUsage

CM_FULL_RESOURCE_DESCRIPTOR structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_FULL_RESOURCE_DESCRIPTOR** structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.

Syntax

```
typedef struct _CM_FULL_RESOURCE_DESCRIPTOR {
  INTERFACE_TYPE           InterfaceType;
  ULONG                    BusNumber;
  CM_PARTIAL_RESOURCE_LIST PartialResourceList;
} CM_FULL_RESOURCE_DESCRIPTOR, *PCM_FULL_RESOURCE_DESCRIPTOR;
```

Members

`InterfaceType`

Specifies the type of bus to which the device is connected. This must be one of the types defined by INTERFACE_TYPE, in Wdm.h or Ntddk.h. (Not used by WDM drivers.)

`BusNumber`

The system-assigned, driver-supplied, zero-based number of the bus to which the device is connected. (Not used by WDM drivers.)

`PartialResourceList`

A CM_PARTIAL_RESOURCE_LIST structure.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_INT13_DRIVE_PARAMETER structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_INT13_DRIVE_PARAMETER** structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a disk controller if the system can collect this information during the boot process.

Syntax

```
typedef struct _CM_INT13_DRIVE_PARAMETER {
  USHORT DriveSelect;
  ULONG  MaxCylinders;
  USHORT SectorsPerTrack;
  USHORT MaxHeads;
  USHORT NumberDrives;
} CM_INT13_DRIVE_PARAMETER, *PCM_INT13_DRIVE_PARAMETER;
```

Members

`DriveSelect`

The drive selected value.

`MaxCylinders`

The maximum number of cylinders.

`SectorsPerTrack`

The number of sectors per track.

`MaxHeads`

The maximum number of heads.

`NumberDrives`

The number of drives.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoQueryDeviceDescription

IoReportResourceUsage

CM_KEYBOARD_DEVICE_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_KEYBOARD_DEVICE_DATA** structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a keyboard peripheral if the system can collect this information during the boot process.

Syntax

```
typedef struct _CM_KEYBOARD_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  UCHAR  Type;
  UCHAR  Subtype;
  USHORT KeyboardFlags;
} CM_KEYBOARD_DEVICE_DATA, *PCM_KEYBOARD_DEVICE_DATA;
```

Members

`Version`

The version number of this structure.

`Revision`

The revision of this structure.

`Type`

The type of the keyboard.

`Subtype`

The subtype of the keyboard.

`KeyboardFlags`

Defined by x86 BIOS INT 16h, function 02 as:

BIT	DEFINED AS
7	Insert on.
6	Caps Lock on.
5	Num Lock on.
4	Scroll Lock on.
3	Alt Key is down.
2	Ctrl Key is down.

1	Left shift key is down.
0	Right shift key is down.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IoQueryDeviceDescription

IoReportResourceUsage

CM_MCA_POS_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_MCA_POS_DATA** structure is obsolete. It defines IBM-compatible MCA POS configuration information for a slot.

Syntax

```
typedef struct _CM_MCA_POS_DATA {
  USHORT AdapterId;
  UCHAR  PosData1;
  UCHAR  PosData2;
  UCHAR  PosData3;
  UCHAR  PosData4;
} CM_MCA_POS_DATA, *PCM_MCA_POS_DATA;
```

Members

`AdapterId`

`PosData1`

`PosData2`

`PosData3`

`PosData4`

Remarks

This structure is used by the obsolete HalGetBusData and HalGetBusDataByOffset routines.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalGetBusData

HalGetBusDataByOffset

CM_PARTIAL_RESOURCE_DESCRIPTOR structure

1/30/2019 • 9 minutes to read • Edit Online

The **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure specifies one or more system hardware resources, of a single type, assigned to a device. This structure is used to create an array within a CM_PARTIAL_RESOURCE_LIST structure.

Syntax

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
  UCHAR  Type;
  UCHAR  ShareDisposition;
  USHORT Flags;
  union {
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Generic;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Port;
    struct {
#if ...
      USHORT    Level;
      USHORT    Group;
#else
      ULONG     Level;
#endif
      ULONG     Vector;
      KAFFINITY Affinity;
    } Interrupt;
    struct {
      union {
        struct {
          USHORT    Group;
          USHORT    Reserved;
          USHORT    MessageCount;
          ULONG     Vector;
          KAFFINITY Affinity;
        } Raw;
        struct {
#if ...
          USHORT    Level;
          USHORT    Group;
#else
          ULONG     Level;
#endif
          ULONG     Vector;
          KAFFINITY Affinity;
        } Translated;
      } DUMMYUNIONNAME;
    } MessageInterrupt;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Memory;
    struct {
      ULONG Channel;
      ULONG Port;
      ULONG Reserved1;
```

```
      ULONG Reserved1;
    } Dma;
    struct {
      ULONG Channel;
      ULONG RequestLine;
      UCHAR TransferWidth;
      UCHAR Reserved1;
      UCHAR Reserved2;
      UCHAR Reserved3;
    } DmaV3;
    struct {
      ULONG Data[3];
    } DevicePrivate;
    struct {
      ULONG Start;
      ULONG Length;
      ULONG Reserved;
    } BusNumber;
    struct {
      ULONG DataSize;
      ULONG Reserved1;
      ULONG Reserved2;
    } DeviceSpecificData;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length40;
    } Memory40;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length48;
    } Memory48;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length64;
    } Memory64;
    struct {
      UCHAR Class;
      UCHAR Type;
      UCHAR Reserved1;
      UCHAR Reserved2;
      ULONG IdLowPart;
      ULONG IdHighPart;
    } Connection;
  } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR, *PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

Members

`Type`

Identifies the resource type. The constant value specified for **Type** indicates which structure within the **u** union is valid, as indicated in the following table. (These flags are used within both **CM_PARTIAL_RESOURCE_DESCRIPTOR** and IO_RESOURCE_DESCRIPTOR structures, except where noted.)

TYPE VALUE	U MEMBER SUBSTRUCTURE
CmResourceTypePort	**u.Port**
CmResourceTypeInterrupt	**u.Interrupt** or **u.MessageInterrupt**.If the CM_RESOURCE_INTERRUPT_MESSAGE flag of **Flags** is set, use **u.MessageInterrupt**; otherwise, use **u.Interrupt**.
CmResourceTypeMemory	**u.Memory**

CmResourceTypeMemoryLarge	One of **u.Memory40**, **u.Memory48**, or **u.Memory64**.The CM_RESOURCE_MEMORY_LARGE_*XXX* flags set in the **Flags** member determines which structure is used.
CmResourceTypeDma	**u.Dma** (if CM_RESOURCE_DMA_V3 is not set) or **u.DmaV3** (if CM_RESOURCE_DMA_V3 flag is set)
CmResourceTypeDevicePrivate	**u.DevicePrivate**
CmResourceTypeBusNumber	**u.BusNumber**
CmResourceTypeDeviceSpecific	**u.DeviceSpecificData**(Not used within **IO_RESOURCE_DESCRIPTOR**.)
CmResourceTypePcCardConfig	**u.DevicePrivate**
CmResourceTypeMfCardConfig	**u.DevicePrivate**
CmResourceTypeConnection	**u.Connection**
CmResourceTypeConfigData	*Reserved for system use.*
CmResourceTypeNonArbitrated	*Not used.*

`ShareDisposition`

Indicates whether the described resource can be shared. Valid constant values are listed in the following table.

VALUE	MEANING
CmResourceShareDeviceExclusive	The device requires exclusive use of the resource.
CmResourceShareDriverExclusive	The driver requires exclusive use of the resource. (Not supported for WDM drivers.)

CmResourceShareShared	The resource can be shared without restriction.

`Flags`

Contains flag bits that are specific to the resource type, as indicated in the following table. Flags can be bitwise-ORed together as appropriate.

RESOURCE TYPE	FLAG	DEFINITION
CmResourceTypePort		
	CM_RESOURCE_PORT_MEMORY	The device is accessed in memory address space.
	CM_RESOURCE_PORT_IO	The device is accessed in I/O address space.
	CM_RESOURCE_PORT_10_BIT_DECODE	The device decodes 10 bits of the port address.
	CM_RESOURCE_PORT_12_BIT_DECODE	The device decodes 12 bits of the port address.
	CM_RESOURCE_PORT_16_BIT_DECODE	The device decodes 16 bits of the port address.
	CM_RESOURCE_PORT_POSITIVE_DECODE	The device uses "positive decode" instead of "subtractive decode". (In general, PCI devices use positive decode and ISA buses use subtractive decode.)
	CM_RESOURCE_PORT_PASSIVE_DECODE	The device decodes the port but the driver does not use it.
	CM_RESOURCE_PORT_WINDOW_DECODE	Reserved for system use.
CmResourceTypeInterrupt		
	CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE	The IRQ line is level-triggered. (These IRQs are usually sharable.)
	CM_RESOURCE_INTERRUPT_LATCHED	The IRQ line is edge-triggered.

	CM_RESOURCE_INTERRUPT_MESSAGE	If this flag is set, the interrupt is a message-signaled interrupt. Otherwise, the interrupt is a line-based interrupt. This flag can be set starting with Windows Vista.
	CM_RESOURCE_INTERRUPT_POLICY_INCLUDED	Not used with the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure. For more information about this flag, see IO_RESOURCE_DESCRIPTOR.
	CM_RESOURCE_INTERRUPT_SECONDARY_INTERRUPT	The interrupt is a secondary interrupt. This flag can be set starting with Windows 8. For more information about secondary interrupts, see GPIO Interrupts.
	CM_RESOURCE_INTERRUPT_WAKE_HINT	The interrupt is capable of waking the operating system from a low-power idle state or a system sleep state. This flag can be set starting with Windows 8. For more information about wake capabilities, see Enabling Device Wake-Up.
CmResourceTypeMemory		
	CM_RESOURCE_MEMORY_READ_WRITE	The memory range is readable and writable.
	CM_RESOURCE_MEMORY_READ_ONLY	The memory range is read-only.
	CM_RESOURCE_MEMORY_WRITE_ONLY	The memory range is write-only.
	CM_RESOURCE_MEMORY_PREFETCHABLE	The memory range is prefetchable.
	CM_RESOURCE_MEMORY_COMBINEDWRITE	Combined-write caching is allowed.
	CM_RESOURCE_MEMORY_24	The device uses 24-bit addressing.
	CM_RESOURCE_MEMORY_CACHEABLE	The memory range is cacheable.
CmResourceTypeMemoryLarge		

	CM_RESOURCE_MEMORY_LARGE_40	The memory descriptor uses the **u.Memory40** member.
	CM_RESOURCE_MEMORY_LARGE_48	The memory descriptor uses the **u.Memory48** member.
	CM_RESOURCE_MEMORY_LARGE_64	The memory descriptor uses the **u.Memory64** member.
CmResourceTypeDma		
	CM_RESOURCE_DMA_8	8-bit DMA channel
	CM_RESOURCE_DMA_16	16-bit DMA channel
	CM_RESOURCE_DMA_32	32-bit DMA channel
	CM_RESOURCE_DMA_8_AND_16	8-bit and 16-bit DMA channel
	CM_RESOURCE_DMA_BUS_MASTER	The device supports bus master DMA transfers.
	CM_RESOURCE_DMA_TYPE_A	Type A DMA
	CM_RESOURCE_DMA_TYPE_B	Type B DMA
	CM_RESOURCE_DMA_TYPE_F	Type F DMA
	CM_RESOURCE_DMA_V3	Use the **DmaV3** member instead of the **Dma** member. The **DmaV3** member is available starting with Windows 8.

`u`

`u.Generic`

`u.Generic.Start`

`u.Generic.Length`

`u.Port`

`u.Port.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Port.Length`

The length, in bytes, of the range of allocated I/O port addresses.

`u.Interrupt`

`u.Interrupt.Level.#1877845754`

`u.Interrupt.Group`

Specifies the processor group number. This member exists only if the NT_PROCESSOR_GROUPS constant is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Group** and **Affinity** members together specify a group affinity that indicates which processors the device can interrupt. To specify an affinity for any group, set **Group** to ALL_PROCESSOR_GROUPS.

`u.Interrupt.Level.#3336307498`

`u.Interrupt.Vector`

For raw resources: Specifies the device's bus-specific interrupt vector (if appropriate for the platform and bus).

For translated resources: Specifies the global system interrupt vector assigned to the device.

For more information about raw and translated resources, see Remarks.

`u.Interrupt.Affinity`

Contains a **KAFFINITY**-typed bitmask value indicating the set of processors the device can interrupt. To indicate that the device can interrupt any processor, this member is set to -1.

`u.MessageInterrupt`

`u.MessageInterrupt.DUMMYUNIONNAME`

`u.MessageInterrupt.DUMMYUNIONNAME.Raw`

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Group`

Specifies a processor group number. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Group** and **Affinity** members together specify a group affinity that indicates which processors can receive the device's interrupts. To specify an affinity for any group, set **Group** to ALL_PROCESSOR_GROUPS.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Reserved`

Not used.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.MessageCount`

Specifies the number of message-signaled interrupts generated for this driver.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Vector`

Specifies the device's interrupt vector.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Affinity`

Specifies a KAFFINITY value that indicates the processors that receive the device's interrupts.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Level.#1877845754`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Group`

Specifies a processor group number. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Translated.Group** and

Translated.Affinity members together specify a group affinity that indicates which processors can receive the device's interrupts. To specify an affinity for any group, set **Translated.Group** to ALL_PROCESSOR_GROUPS.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Level.#3336307498`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Vector`

Specifies the device's interrupt vector.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Affinity`

Specifies a KAFFINITY value that identifies the processors that receive the device's interrupts.

`u.Memory`

`u.Memory.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory.Length`

Specifies the length, in bytes, of the range of allocated memory addresses.

`u.Dma`

`u.Dma.Channel`

Specifies the number of the DMA channel on a system DMA controller that the device can use.

`u.Dma.Port`

Specifies the number of the DMA port that an MCA-type device can use.

`u.Dma.Reserved1`

Not used.

`u.DmaV3`

`u.DmaV3.Channel`

Specifies the number of the DMA channel on the system DMA controller that is allocated to the device.

`u.DmaV3.RequestLine`

Specifies the number of the request line on the system DMA controller that is allocated to the device.

`u.DmaV3.TransferWidth`

Specifies the width, in bits, of the data bus that the system DMA controller that is allocated to the device uses to transfer data to or from the device.

`u.DmaV3.Reserved1`

Not used.

`u.DmaV3.Reserved2`

Not used.

`u.DmaV3.Reserved3`

Not used.

`u.DevicePrivate`

`u.DevicePrivate.Data`

`u.BusNumber`

`u.BusNumber.Start`

Specifies the lowest-numbered of a range of contiguous buses allocated to the device.

`u.BusNumber.Length`

Specifies the number of buses allocated to the device.

`u.BusNumber.Reserved`

Not used.

`u.DeviceSpecificData`

`u.DeviceSpecificData.DataSize`

Specifies the number of bytes appended to the end of the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure.

`u.DeviceSpecificData.Reserved1`

Not used.

`u.DeviceSpecificData.Reserved2`

Not used.

`u.Memory40`

`u.Memory40.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory40.Length40`

Contains the high 32 bits of the 40-bit length, in bytes, of the range of allocated memory addresses. The lowest 8 bits are treated as zero.

`u.Memory48`

`u.Memory48.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory48.Length48`

Contains the high 32 bits of the 48-bit length, in bytes, of the range of allocated memory addresses. The lowest 16 bits are treated as zero.

`u.Memory64`

`u.Memory64.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory64.Length64`

Contains the high 32 bits of the 64-bit length, in bytes, of the range of allocated memory addresses. The lowest 32 bits are treated as zero.

`u.Connection`

`u.Connection.Class`

Specifies the connection class. This member is set to one of the following values.

VALUE	MEANING
CM_RESOURCE_CONNECTION_CLASS_GPIO	Access the device through one or more pins on a GPIO controller.
CM_RESOURCE_CONNECTION_CLASS_SERIAL	Access the device through a serial bus or serial port.

`u.Connection.Type`

Specifies the connection type.

If **Class** = CM_RESOURCE_CONNECTION_CLASS_GPIO, **Type** is set to the following value.

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_GPIO_IO	Access the device through GPIO pins that are configured for I/O.

Note A GPIO pin that is configured as an interrupt request input is accessed as an ordinary interrupt resource (**CmResourceTypeInterrupt**).

If **Class** = CM_RESOURCE_CONNECTION_CLASS_SERIAL, **Type** is set to one of the following values.

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_SERIAL_I2C	The device is connected to an I^2C bus.
CM_RESOURCE_CONNECTION_TYPE_SERIAL_SPI	The device is connected to an SPI bus.

CM_RESOURCE_CONNECTION_TYPE_SERIAL_UART	The device is connected to a serial port.

`u.Connection.Reserved1`

Not used.

`u.Connection.Reserved2`

Not used.

`u.Connection.IdLowPart`

Contains the lower 32 bits of the 64-bit connection ID.

`u.Connection.IdHighPart`

Contains the upper 32 bits of the 64-bit connection ID.

Remarks

A **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure can describe either a raw (bus-relative) resource or a translated (system physical) resource, depending on the routine or IRP with which it is being used. For more information, see Raw and Translated Resources and IRP_MN_START_DEVICE.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FLOPPY_DEVICE_DATA

CM_FULL_RESOURCE_DESCRIPTOR

CM_KEYBOARD_DEVICE_DATA

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_SCSI_DEVICE_DATA

CM_SERIAL_DEVICE_DATA

DMA_OPERATIONS

IO_RESOURCE_DESCRIPTOR

IRP_MN_START_DEVICE

IoConnectInterrupt

IoGetDeviceProperty

IoReportResourceForDetection

KAFFINITY

RtlCmDecodeMemIoResource

RtlCmEncodeMemIoResource

CM_PARTIAL_RESOURCE_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_PARTIAL_RESOURCE_LIST** structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.

Syntax

```
typedef struct _CM_PARTIAL_RESOURCE_LIST {
  USHORT                          Version;
  USHORT                          Revision;
  ULONG                           Count;
  CM_PARTIAL_RESOURCE_DESCRIPTOR PartialDescriptors[1];
} CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;
```

Members

`Version`

The version number of this structure. This value should be 1.

`Revision`

The revision of this structure. This value should be 1.

`Count`

The number of elements contained in the **PartialDescriptors** array.

`PartialDescriptors`

The first element in an array of one or more CM_PARTIAL_RESOURCE_DESCRIPTOR structures.

Remarks

This structure is the header for an array of **CM_PARTIAL_RESOURCE_DESCRIPTOR** structures. The **PartialDescriptors** member contains the first element in this array, and the **Count** member specifies the total number of array elements. If the array contains more than one element, the remaining elements in the array immediately follow the **CM_PARTIAL_RESOURCE_LIST** structure in memory. The total number of bytes occupied by the **CM_PARTIAL_RESOURCE_LIST** structure and any array elements that follow this structure is **sizeof**(**CM_PARTIAL_RESOURCE_LIST**) + (**Count** - 1) * **sizeof**(**CM_PARTIAL_RESOURCE_DESCRIPTOR**).

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FULL_RESOURCE_DESCRIPTOR

CM_RESOURCE_LIST structure

1/30/2019 • 3 minutes to read • Edit Online

The **CM_RESOURCE_LIST** structure specifies all of the system hardware resources assigned to a device.

Syntax

```
typedef struct _CM_RESOURCE_LIST {
  ULONG                    Count;
  CM_FULL_RESOURCE_DESCRIPTOR List[1];
} CM_RESOURCE_LIST, *PCM_RESOURCE_LIST;
```

Members

`Count`

The number of full resource descriptors that are specified by this **CM_RESOURCE_LIST** structure. The **List** member is the header for the first full resource descriptor. For WDM drivers, **Count** is always 1.

`List`

The CM_FULL_RESOURCE_DESCRIPTOR structure that serves as the header for the first full resource descriptor. If the **CM_RESOURCE_LIST** structure contains more than one full resource descriptor, the second full resource descriptor immediately follows the first in memory, and so on. The size of each full resource descriptor depends on the length of the CM_PARTIAL_RESOURCE_DESCRIPTOR array that it contains. For more information, see the following Remarks section.

Remarks

This structure describes the assignment of hardware resources to a device. An IRP_MN_START_DEVICE IRP uses this structure to specify the resources that the Plug and Play manager assigns to a device. Drivers for legacy devices use this structure to pass their resource requirements to the IoReportResourceForDetection routine. For more information about hardware resource allocation, see Hardware Resources.

The **CM_RESOURCE_LIST** structure is a header for a larger data structure, of variable size, that contains one or more full resource descriptors. All of the data in this larger structure occupies a contiguous block of memory. Each full resource descriptor occupies a subblock within the larger block.

A full resource descriptor begins with a CM_FULL_RESOURCE_DESCRIPTOR structure, which serves as a header for an array of CM_PARTIAL_RESOURCE_DESCRIPTOR structures. The length of this array determines the size of the full resource descriptor. The last member in the **CM_FULL_RESOURCE_DESCRIPTOR** structure is a CM_PARTIAL_RESOURCE_LIST structure that contains, as its last member, the first element in this array. If the array contains more than one element, the remaining elements immediately follow, in memory, the end of the **CM_PARTIAL_RESOURCE_LIST** structure, which is also the end of the **CM_FULL_RESOURCE_DESCRIPTOR** structure.

Driver code can use pointer arithmetic to step from one full resource descriptor to the next. For example, if a parameter named *list* is a pointer to the **CM_FULL_RESOURCE_DESCRIPTOR** structure at the start of one full resource descriptor, *list* can be updated to point to the start of the next full resource descriptor as follows:

```
    list = (PCM_FULL_RESOURCE_DESCRIPTOR)(list->PartialResourceList.PartialDescriptors +
                                          list->PartialResourceList.Count);
```

In this example, `list->PartialResourceList.PartialDescriptors` is a pointer to the start of the **CM_PARTIAL_RESOURCE_DESCRIPTOR** array, and `list->PartialResourceList.Count` is the number of elements in the array. For more information about the **PartialDescriptors** and **Count** members, see CM_PARTIAL_RESOURCE_LIST. #### Examples All PnP drivers must handle IRP_MN_START_DEVICE IRPs. Typically, a driver's handler for this IRP walks the lists of assigned resources that are pointed to by the **Parameters.StartDevice.AllocatedResources** and **Parameters.StartDevice.AllocatedResourcesTranslated** members of the IO_STACK_LOCATION structure in the IRP. The following code example contains a function—named GetAssignedResources—that is called in the handler to walk each list. This function verifies that the required resources are specified in the list, and configures the device to use the resources. The GetAssignedResources function returns **TRUE** if it succeeds. Otherwise, it returns **FALSE** (probably from the **switch** statement, although the details are omitted to simplify the code example).

```
/* Process the assigned hardware resources. */

BOOLEAN GetAssignedResources(PCM_RESOURCE_LIST reslist)
{
    PCM_FULL_RESOURCE_DESCRIPTOR list;

    list = reslist->List;

    for (int ix = 0; ix < reslist->Count; ++ix)
    {
        /* Process resources in CM_FULL_RESOURCE_DESCRIPTOR block number ix. */

        for (int jx = 0; jx < list->PartialResourceList.Count; ++jx)
        {
            PCM_PARTIAL_RESOURCE_DESCRIPTOR desc;

            desc = list->PartialResourceList.PartialDescriptors + jx;

            switch (desc->Type)
            {
                /* Process element jx in PartialDescriptors array. */

                ...

            }
        }

        /* Advance to next CM_FULL_RESOURCE_DESCRIPTOR block in memory. */

        list = (PCM_FULL_RESOURCE_DESCRIPTOR)(list->PartialResourceList.PartialDescriptors +
                                              list->PartialResourceList.Count);
    }

    return TRUE;
}
```

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FLOPPY_DEVICE_DATA

CM_FULL_RESOURCE_DESCRIPTOR

CM_KEYBOARD_DEVICE_DATA

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_SCSI_DEVICE_DATA

CM_SERIAL_DEVICE_DATA

IRP_MN_START_DEVICE

IoConnectInterrupt

IoGetDeviceProperty

IoReportResourceForDetection

CM_SCSI_DEVICE_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_SCSI_DEVICE_DATA** structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a SCSI HBA if the system can collect this information during the boot process.

Syntax

```
typedef struct _CM_SCSI_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  UCHAR  HostIdentifier;
} CM_SCSI_DEVICE_DATA, *PCM_SCSI_DEVICE_DATA;
```

Members

`Version`

The version number of this structure.

`Revision`

The revision for this structure.

`HostIdentifier`

The SCSI bus identifier used by the ARC firmware.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IoQueryDeviceDescription

IoReportResourceUsage

CM_SERIAL_DEVICE_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_SERIAL_DEVICE_DATA** structure defines a device-type-specific data record that is stored in the \Registry\Machine\Hardware\Description tree for a serial controller if the system can collect this information during the boot process.

Syntax

```
typedef struct _CM_SERIAL_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  ULONG  BaudClock;
} CM_SERIAL_DEVICE_DATA, *PCM_SERIAL_DEVICE_DATA;
```

Members

`Version`

The version number of this structure.

`Revision`

The revision of this structure.

`BaudClock`

The clock baud rate, in megahertz, at which data is transferred.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IoQueryDeviceDescription

IoReportResourceUsage

COUNTED_REASON_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **COUNTED_REASON_CONTEXT** structure contains one or more strings that give reasons for a power request.

Syntax

```
typedef struct _COUNTED_REASON_CONTEXT {
  ULONG Version;
  ULONG Flags;
  union {
    struct {
      UNICODE_STRING  ResourceFileName;
      USHORT          ResourceReasonId;
      ULONG           StringCount;
      PUNICODE_STRING ReasonStrings;
    } DUMMYSTRUCTNAME;
    UNICODE_STRING SimpleString;
  } DUMMYUNIONNAME;
} COUNTED_REASON_CONTEXT, *PCOUNTED_REASON_CONTEXT;
```

Members

`Version`

The version number of the structure. Set this member to DIAGNOSTIC_REASON_VERSION.

`Flags`

Indicates whether the structure contains a simple reason string or a detailed set of reason strings. Set this member to one of the following constants:

- DIAGNOSTIC_REASON_SIMPLE_STRING
- DIAGNOSTIC_REASON_DETAILED_STRING

If **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING, the **SimpleString** member of the union is valid. If **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING, the **ResourceFileName**, **ResourceReasonId**, **StringCount**, and **ReasonStrings** members are valid (and the **SimpleString** member is not valid).

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ResourceFileName`

A pointer to a wide-character, null-terminated string that contains the pathname of a resource file. This resource file contains one or more localized strings that give reasons for a power request. This member is optional and can be specified as **NULL** or as an empty string if no resource file is required. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ResourceReasonId`

The resource ID assigned to the first reason string in the resource file that is specified by **ResourceFileName**. This

member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.StringCount`

The number of reason strings in the **ReasonStrings** array or in the resource file that is specified by **ResourceFileName**. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReasonStrings`

A pointer to an array of string pointers. Each array element is a pointer to a wide-character, null-terminated string. The number of array elements is specified by **StringCount**. This member is valid only if **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING.

`DUMMYUNIONNAME.SimpleString`

A pointer to a wide-character, null-terminated string that explains the reason for a power request. This member is valid only if **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING.

Remarks

This structure is used by the PoCreatePowerRequest routine.

The power manager uses the reason string or strings contained in this structure as a diagnostic aid during functional and performance testing.

The **COUNTED_REASON_CONTEXT** structure can contain either a simple reason string or a set of detailed reason strings. If **Flags** = DIAGNOSTIC_REASON_SIMPLE_STRING, the **SimpleString** member points to a string that explains the reason for the power request. If **Flags** = DIAGNOSTIC_REASON_DETAILED_STRING, the **ResourceFileName**, **ResourceReasonId**, **StringCount**, and **ReasonStrings** members can give a detailed set of reasons for the power request.

The DIAGNOSTIC_REASON_DETAILED_STRING flag supports localization. If the localized resource file specified by **ResourceFileName** exists, the power manager retrieves a set of resource strings from the file and uses these strings in place of the strings contained in the **ReasonStrings** array. **ResourceId** specifies the resource index of the first string in the file to use as a reason string, and **StringCount** specifies the number of resource strings to use as reason strings. The resource strings must have consecutive resource indexes.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

PoCreatePowerRequest

D3COLD_LAST_TRANSITION_STATUS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **D3COLD_LAST_TRANSITION_STATUS** enumeration indicates whether the most recent transition to the D3hot device power state was followed by a transition to the D3cold device power state.

Syntax

```
typedef enum _D3COLD_LAST_TRANSITION_STATUS {
  LastDStateTransitionStatusUnknown,
  LastDStateTransitionD3hot,
  LastDStateTransitionD3cold
} D3COLD_LAST_TRANSITION_STATUS, *PD3COLD_LAST_TRANSITION_STATUS;
```

Constants

LastDStateTransitionStatusUnknown	Information is not available from the bus driver or platform firmware about whether the most recent transition to D3hot was followed by a transition to D3cold.
LastDStateTransitionD3hot	The most recent transition to the D3hot device power state was not followed by a transition to the D3cold device power state.
LastDStateTransitionD3cold	The most recent transition to the D3hot device power state was followed by a transition to the D3cold device power state.

Remarks

The *LastTransitionStatus* parameter of the GetLastTransitionStatus routine is a pointer to a variable of type **D3COLD_LAST_TRANSITION_STATUS**.

Requirements

Header	wdm.h (include Wdm.h)

See Also

GetLastTransitionStatus

D3COLD_SUPPORT_INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **D3COLD_SUPPORT_INTERFACE** interface structure contains pointers to the routines in the GUID_D3COLD_SUPPORT_INTERFACE driver interface.

Syntax

```
typedef struct _D3COLD_SUPPORT_INTERFACE {
  USHORT                              Size;
  USHORT                              Version;
  PVOID                               Context;
  PINTERFACE_REFERENCE                InterfaceReference;
  PINTERFACE_DEREFERENCE              InterfaceDereference;
  PSET_D3COLD_SUPPORT                 SetD3ColdSupport;
  PGET_IDLE_WAKE_INFO                 GetIdleWakeInfo;
  PGET_D3COLD_CAPABILITY              GetD3ColdCapability;
  PGET_D3COLD_CAPABILITY              GetBusDriverD3ColdSupport;
  PGET_D3COLD_LAST_TRANSITION_STATUS GetLastTransitionStatus;
} D3COLD_SUPPORT_INTERFACE, *PD3COLD_SUPPORT_INTERFACE;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined interface version. The current version of this interface is D3COLD_SUPPORT_INTERFACE_VERSION.

`Context`

A pointer to interface-specific context information.

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the interface's reference count.

`SetD3ColdSupport`

A pointer to a SetD3ColdSupport routine that enables or disables transitions to the D3cold device power state.

`GetIdleWakeInfo`

A pointer to a GetIdleWakeInfo routine that the device driver calls to discover the device power states from which this device can signal wake events to the processor.

`GetD3ColdCapability`

A pointer to a GetD3ColdCapability routine that reports whether this device is capable of entering the D3cold

device power state.

`GetBusDriverD3ColdSupport`

A pointer to a GetBusDriverD3ColdSupport routine that reports whether the underlying bus driver and ACPI system firmware support D3cold for this device.

`GetLastTransitionStatus`

A pointer to a GetLastTransitionStatus routine that reports whether this device's most recent transition to D3hot was followed by a transition to D3cold.

Remarks

A device driver that successfully queries for the GUID_D3COLD_SUPPORT_INTERFACE interface receives a pointer to a **D3COLD_SUPPORT_INTERFACE** structure in which the pointers to the routines in the interface are all non-NULL and valid.

The **D3COLD_SUPPORT_INTERFACE** structure is an extended version of the INTERFACE structure.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h)

See Also

GetBusDriverD3ColdSupport

GetD3ColdCapability

GetIdleWakeInfo

GetLastTransitionStatus

INTERFACE

InterfaceDereference

InterfaceReference

SetD3ColdSupport

DEVICE_CAPABILITIES structure

1/30/2019 • 8 minutes to read • Edit Online

A **DEVICE_CAPABILITIES** structure describes PnP and power capabilities of a device. This structure is returned in response to an IRP_MN_QUERY_CAPABILITIES IRP.

Syntax

```
typedef struct _DEVICE_CAPABILITIES {
  USHORT             Size;
  USHORT             Version;
  ULONG              DeviceD1 : 1;
  ULONG              DeviceD2 : 1;
  ULONG              LockSupported : 1;
  ULONG              EjectSupported : 1;
  ULONG              Removable : 1;
  ULONG              DockDevice : 1;
  ULONG              UniqueID : 1;
  ULONG              SilentInstall : 1;
  ULONG              RawDeviceOK : 1;
  ULONG              SurpriseRemovalOK : 1;
  ULONG              WakeFromD0 : 1;
  ULONG              WakeFromD1 : 1;
  ULONG              WakeFromD2 : 1;
  ULONG              WakeFromD3 : 1;
  ULONG              HardwareDisabled : 1;
  ULONG              NonDynamic : 1;
  ULONG              WarmEjectSupported : 1;
  ULONG              NoDisplayInUI : 1;
  ULONG              Reserved1 : 1;
  ULONG              WakeFromInterrupt : 1;
  ULONG              SecureDevice : 1;
  ULONG              ChildOfVgaEnabledBridge : 1;
  ULONG              DecodeIoOnBoot : 1;
  ULONG              Reserved : 9;
  ULONG              Address;
  ULONG              UINumber;
  DEVICE_POWER_STATE DeviceState[POWER_SYSTEM_MAXIMUM];
  SYSTEM_POWER_STATE SystemWake;
  DEVICE_POWER_STATE DeviceWake;
  ULONG              D1Latency;
  ULONG              D2Latency;
  ULONG              D3Latency;
} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;
```

Members

`Size`

Specifies the size of the structure, in bytes. This field is set by the component that sends the **IRP_MN_QUERY_CAPABILITIES** request.

`Version`

Specifies the version of the structure, currently version 1. This field is set by the component that sends the **IRP_MN_QUERY_CAPABILITIES** request.

`DeviceD1`

Specifies whether the device hardware supports the D1 power state. Drivers should not change this value.

`DeviceD2`

Specifies whether the device hardware supports the D2 power state. Drivers should not change this value.

`LockSupported`

Specifies whether the device supports physical-device locking that prevents device ejection. This member pertains to ejecting the device from its slot, rather than ejecting a piece of removable media from the device.

`EjectSupported`

Specifies whether the device supports software-controlled device ejection while the system is in the **PowerSystemWorking** state. This member pertains to ejecting the device from its slot, rather than ejecting a piece of removable media from the device.

`Removable`

Specifies whether the device can be dynamically removed from its immediate parent. If **Removable** is set to **TRUE**, the device does not belong to the same physical object as its parent.

For example, if **Removable** is set to **TRUE** for a USB composite device inside a multifunction printer, the composite device does not belong to the physical object of its immediate parent, such as a USB hub inside a notebook PC.

In most cases the bus driver, not the function driver, should determine the value of the**Removable** parameter of the device. For USB devices, the USB hub driver sets the **Removable** parameter. It should not be modified by the function driver.

If **Removable** is set to **TRUE**, the device is displayed in the **Unplug or Eject Hardware** program, unless **SurpriseRemovalOK** is also set to **TRUE**.

`DockDevice`

Specifies whether the device is a docking peripheral.

`UniqueID`

Specifies whether the device's instance ID is unique system-wide. This bit is clear if the instance ID is unique only within the scope of the bus. For more information, see Device Identification Strings.

`SilentInstall`

Specifies whether Device Manager should suppress all installation dialog boxes; except required dialog boxes such as "no compatible drivers found."

`RawDeviceOK`

Specifies whether the driver for the underlying bus can drive the device if there is no function driver (for example, SCSI devices in pass-through mode). This mode of operation is called raw mode.

`SurpriseRemovalOK`

Specifies whether the function driver for the device can handle the case where the device is removed before Windows can send **IRP_MN_QUERY_REMOVE_DEVICE** to it. If **SurpriseRemovalOK** is set to **TRUE**, the device can be safely removed from its immediate parent regardless of the state that its driver is in.

For example, a standard USB mouse does not maintain any state in its hardware and thus can be safely removed at any time. However, an external hard disk whose driver caches writes in memory cannot be safely removed without

first letting the driver flush its cache to the hardware.

> **Note** Drivers for USB devices that support surprise removal must set this to **TRUE** only when the IRP is being passed back up the driver stack.

`WakeFromD0`

Specifies whether the device can respond to an external wake signal while in the D0 state. Drivers should not change this value.

`WakeFromD1`

Specifies whether the device can respond to an external wake signal while in the D1 state. Drivers should not change this value.

`WakeFromD2`

Specifies whether the device can respond to an external wake signal while in the D2 state. Drivers should not change this value.

`WakeFromD3`

Specifies whether the device can respond to an external wake signal while in the D3 state. Drivers should not change this value.

`HardwareDisabled`

When set, this flag specifies that the device's hardware is disabled.

A device's parent bus driver or a bus filter driver sets this flag when such a driver determines that the device hardware is disabled.

The PnP manager sends one **IRP_MN_QUERY_CAPABILITIES** IRP right after a device is enumerated and sends another after the device has been started. The PnP manager only checks this bit right after the device is enumerated. Once the device is started, this bit is ignored.

`NonDynamic`

Reserved for future use.

`WarmEjectSupported`

Reserved for future use.

`NoDisplayInUI`

Do not display the device in the user interface. If this bit is set, the device is never displayed in the user interface, even if the device is present but fails to start. Only bus drivers and associated bus filter drivers should set this bit. (Also see the **PNP_DEVICE_DONT_DISPLAY_IN_UI** flag in the PNP_DEVICE_STATE structure.)

`Reserved1`

Reserved for system use.

`WakeFromInterrupt`

Indicates whether the driver or ACPI is responsible for handling the wake event. If set, the driver is responsible for handling the wake event. ACPI arms the device when it receives an IRP_MN_WAIT_WAKE IRP, but does not connect the interrupt, complete the IRP to notify the device stack of a wake event.

`SecureDevice`

Indicates whether the device is a secure device.

`ChildOfVgaEnabledBridge`

For a VGA device, indicates whether the parent bridge has the VGA decoding bit set.

`DecodeIoOnBoot`

Indictates whether the device has IO decode enabled on boot.

`Reserved`

Reserved for system use.

`Address`

Specifies an address indicating where the device is located on its underlying bus.

The interpretation of this number is bus-specific. If the address is unknown or the bus driver does not support an address, the bus driver leaves this member at its default value of 0xFFFFFFFF.

The following list describes the information certain bus drivers store in the **Address** field for their child devices:

1394
Does not supply an address because the addresses are volatile. Defaults to 0xFFFFFFFF.

EISA
Slot Number (0-F).

IDE
For an IDE device, the address contains the target ID and LUN. For an IDE channel, the address is zero or one (0 = primary channel and 1 = secondary channel).

ISApnp
Does not supply an address. Defaults to 0xFFFFFFFF.

PC Card (PCMCIA)
The socket number (typically 0x00 or 0x40).

PCI
The device number in the high word and the function number in the low word.

SCSI
The target ID.

USB
The port number.

`UINumber`

Specifies a number associated with the device that can be displayed in the user interface.

This number is typically a user-perceived slot number, such as a number printed next to the slot on the board, or some other number that makes locating the physical device easier for the user. For buses with no such convention, or when the **UINumber** is unknown, the bus driver leaves this member at its default value of 0xFFFFFFFF.

`DeviceState`

An array of values indicating the most-powered device power state that the device can maintain for each system power state. The **DeviceState[PowerSystemWorking]** element of the array corresponds to the S0 system state. The entry for **PowerSystemUnspecified** is reserved for system use.

The entries in this array are based on the capabilities of the parent devnode. As a general rule, a driver should not

change these values. However, if necessary, a driver can lower the value, for example, from **PowerDeviceD1** to **PowerDeviceD2**.

If the bus driver is unable to determine the appropriate device power state for a root-enumerated device, it sets **DeviceState[PowerSystemWorking]** to **PowerDeviceD0** and all other entries to **PowerDeviceD3**.

`SystemWake`

Specifies the least-powered system power state from which the device can signal a wake event. A value of **PowerSystemUnspecified** indicates that the device cannot wake the system.

A bus driver can get this information from its parent devnode.

In general, a driver should not change this value. If necessary, however, a driver can raise the power state, for example, from **PowerSystemHibernate** to **PowerSystemS1**, to indicate that its device cannot wake the system from a hibernation state but can from a higher-powered sleep state.

`DeviceWake`

Specifies the least-powered device power state from which the device can signal a wake event. A value of **PowerDeviceUnspecified** indicates that the device cannot signal a wake event.

`D1Latency`

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD1** state. Set to zero if the device does not support the D1 state.

`D2Latency`

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD2** state. Set to zero if the device does not support the D2 state.

`D3Latency`

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD3** state. Set to zero if the device does not support the D3 state.

Remarks

Bus drivers set the appropriate values in this structure in response to an **IRP_MN_QUERY_CAPABILITIES** IRP. Bus filter drivers, function drivers, and filter drivers might alter the capabilities set by the bus driver.

Drivers that send an **IRP_MN_QUERY_CAPABILITIES** request must initialize the **Size**, **Version**, **Address**, and **UINumber** members of this structure before sending the IRP.

For more information about using the **DEVICE_CAPABILITIES** structure to describe a device's power capabilities, see Reporting Device Power Capabilities.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IRP_MN_QUERY_CAPABILITIES

PNP_DEVICE_STATE

DEVICE_DESCRIPTION structure

1/30/2019 • 7 minutes to read • Edit Online

The **DEVICE_DESCRIPTION** structure describes the attributes of the physical device for which a driver is requesting a DMA adapter.

Syntax

```
typedef struct _DEVICE_DESCRIPTION {
  ULONG            Version;
  BOOLEAN          Master;
  BOOLEAN          ScatterGather;
  BOOLEAN          DemandMode;
  BOOLEAN          AutoInitialize;
  BOOLEAN          Dma32BitAddresses;
  BOOLEAN          IgnoreCount;
  BOOLEAN          Reserved1;
  BOOLEAN          Dma64BitAddresses;
  ULONG            BusNumber;
  ULONG            DmaChannel;
  INTERFACE_TYPE   InterfaceType;
  DMA_WIDTH        DmaWidth;
  DMA_SPEED        DmaSpeed;
  ULONG            MaximumLength;
  ULONG            DmaPort;
  ULONG            DmaAddressWidth;
  ULONG            DmaControllerInstance;
  ULONG            DmaRequestLine;
  PHYSICAL_ADDRESS DeviceAddress;
} DEVICE_DESCRIPTION, *PDEVICE_DESCRIPTION;
```

Members

`Version`

The version of this structure. The **Version** member of the **DEVICE_DESCRIPTION** structure that is passed to the IoGetDmaAdapter routine determines which version of the DMA_ADAPTER structure is returned by this routine. The following is a list of the possible values of the **Version** member and the corresponding **DMA_ADAPTER** versions:

DEVICE_DESCRIPTION_VERSION

If **Version** = **DEVICE_DESCRIPTION_VERSION**, **IoGetDmaAdapter** ignores the **IgnoreCount** member, and returns version 1 of the **DMA_ADAPTER** structure.

DEVICE_DESCRIPTION_VERSION1

If **Version** = **DEVICE_DESCRIPTION_VERSION1**, **IoGetDmaAdapter** uses the **IgnoreCount** member, and returns version 1 of the **DMA_ADAPTER** structure.

DEVICE_DESCRIPTION_VERSION2

If **Version** = **DEVICE_DESCRIPTION_VERSION2**, **IoGetDmaAdapter** uses the **IgnoreCount** member, and returns version 2 of the **DMA_ADAPTER** structure. Version 2 is available starting with Windows XP.

DEVICE_DESCRIPTION_VERSION3

If **Version** = **DEVICE_DESCRIPTION_VERSION3**, **IoGetDmaAdapter** uses the **IgnoreCount** member, and returns version 3 of the **DMA_ADAPTER** structure. Version 3 is available starting with Windows 8.

`Master`

Whether the device is a bus-master DMA device. Set to **TRUE** if the device is a bus-master DMA device. Set to **FALSE** if it is a subordinate DMA device.

`ScatterGather`

For a bus-master DMA device, this member indicates whether the device supports scatter/gather DMA. Set to **TRUE** if the device can do scatter/gather DMA. Otherwise, set this member to **FALSE**.

For a subordinate DMA device, the **ScatterGather** value is not used. Instead, **IoGetDmaAdapter** assumes that the scatter/gather capability of a subordinate DMA device is the same as that of the underlying system DMA controller to which the device is connected.

`DemandMode`

This member is used only if **Version** is DEVICE_DESCRIPTION_VERSION2.

For a subordinate DMA device, this member indicates whether to use the system DMA controller's demand mode. Set to **TRUE** to use demand mode. Otherwise, set this member to **FALSE**.

For a bus-master DMA device, the **DemandMode** value is not used.

If **Version** is DEVICE_DESCRIPTION_VERSION, DEVICE_DESCRIPTION_VERSION1, or DEVICE_DESCRIPTION_VERSION3, the **DemandMode** value is not used.

`AutoInitialize`

For a subordinate DMA device, this member indicates whether to use the system DMA controller's autoinitialize mode. Set to **TRUE** to use autoinitialize mode. Otherwise, set this member to **FALSE**.

For a bus-master DMA device, the **AutoInitialize** value is not used.

`Dma32BitAddresses`

This member is used only if **Version** is DEVICE_DESCRIPTION_VERSION, DEVICE_DESCRIPTION_VERSION1, or DEVICE_DESCRIPTION_VERSION2.

Dma32BitAddresses specifies whether the device can use full 32-bit addresses for DMA operations. Set to **TRUE** if the device supports 32-bit addresses. Otherwise, set this member to **FALSE**.

If **Version** = DEVICE_DESCRIPTION_VERSION3, the **Dma32BitAddresses** value is not used.

`IgnoreCount`

Whether to ignore the DMA controller's transfer counter. Set to**TRUE** if the DMA controller in this platform does not maintain an accurate transfer counter, and therefore requires a workaround. Otherwise, set this member to **FALSE**.

If **Version** = DEVICE_DESCRIPTION_VERSION, the **IgnoreCount** value is not used.

For more information, see the Remarks section.

`Reserved1`

Reserved for system use. Must be **FALSE**.

`Dma64BitAddresses`

This member is used only if **Version** is DEVICE_DESCRIPTION_VERSION, DEVICE_DESCRIPTION_VERSION1, or DEVICE_DESCRIPTION_VERSION2.

Dma64BitAddresses specifies whether the device can use full 64-bit addresses for DMA operations. Set to **TRUE**

if the device supports 64-bit addresses. Otherwise, set this member to **FALSE**.

If **Version** = DEVICE_DESCRIPTION_VERSION3, the **Dma64BitAddresses** value is not used.

`BusNumber`

The system-assigned bus number for the I/O bus. This member is not used by WDM drivers.

`DmaChannel`

The number of the DMA channel to which a subordinate device is assigned. The device driver obtains this channel number from the resource list that it receives in the IRP_MN_START_DEVICE request that starts the device. For more information about this number, see the description of the **Dma.Channel** member in CM_PARTIAL_RESOURCE_DESCRIPTOR.

`InterfaceType`

The interface type of the I/O bus to use for DMA. Set this member to the INTERFACE_TYPE enumeration value that indicates the interface type. For more information, see the Remarks section.

`DmaWidth`

For a subordinate DMA device, this member specifies the DMA data width for transfers by the system DMA controller. Possible values are **Width8Bits**, **Width16Bits**, **Width32Bits**, and **Width64Bits**.

For a bus-master DMA device, the **DmaWidth** value is not used.

`DmaSpeed`

This member is used only if **Version** is DEVICE_DESCRIPTION_VERSION, DEVICE_DESCRIPTION_VERSION1, or DEVICE_DESCRIPTION_VERSION2.

For a subordinate DMA device, this member specifies one of the following speeds for system DMA: **Compatible**, **TypeA**, **TypeB**, **TypeC**, or **TypeF**.

For a bus-master DMA device, the **DmaSpeed** value is not used.

If **Version** = DEVICE_DESCRIPTION_VERSION3, the **DmaSpeed** value is not used.

`MaximumLength`

The maximum number of bytes the device can transfer in a DMA operation that uses the allocated adapter object.

`DmaPort`

The Microchannel-type bus port number. This parameter is obsolete, but is retained in the structure for compatibility with legacy drivers.

`DmaAddressWidth`

This member is used only if **Version** = DEVICE_DESCRIPTION_VERSION3.

For a bus-master DMA device, **DmaAddressWidth** specifies the width, in bits, of a DMA address. The **DmaAddressWidth** value must be nonzero and must not exceed 64. If the memory address width is greater than the DMA address width, map registers are required to access a region of memory that is beyond the address reach of the DMA controller.

For a subordinate DMA device, the **DmaAddressWidth** value is not used. Instead, **IoGetDmaAdapter** assumes that the address width of a subordinate DMA device is the same as that of the underlying system DMA controller to which the device is connected.

`DmaControllerInstance`

Not used.

`DmaRequestLine`

This member is used only if **Version** = DEVICE_DESCRIPTION_VERSION3.

For a subordinate DMA device, **DmaRequestLine** specifies the request line on the DMA controller to which the device is connected. The device driver obtains the number of this request line from the resource list it receives in the **IRP_MN_START_DEVICE** request that starts the device. For more information about the request line number, see the description of the **u.DmaV3.RequestLine** member in CM_PARTIAL_RESOURCE_DESCRIPTOR.

For a bus-master DMA device, the **DmaRequestLine** value is not used.

`DeviceAddress`

This member is used only if **Version** = DEVICE_DESCRIPTION_VERSION3.

For a subordinate DMA device, **DeviceAddress** is the memory-mapped address of the data register on the device that is used as the source or destination for a DMA transfer. This data register is located at a known, device-specific offset from the device start address. The width of this register is specified by the **DmaWidth** member. The device driver obtains the device start address from the resource list it receives in the **IRP_MN_START_DEVICE** request that starts the device. For more information about this address, see the description of the **u.Memory.Start** member in CM_PARTIAL_RESOURCE_DESCRIPTOR.

For a bus-master DMA device, the **DeviceAddress** member is not used.

Remarks

The driver of a device that uses DMA to transfer data uses the **DEVICE_DESCRIPTION** structure to pass information about the device to the IoGetDmaAdapter routine. The driver calls this routine to request an adapter object for a physical device object (PDO). This PDO represents the device's physical connection to the I/O bus to use for DMA. For more information, see Getting an Adapter Object.

To allocate resources for a DMA controller, the I/O manager needs information about the controller but can obtain some of this information only from a driver. For example, the driver for a bus-master device knows whether the device supports scatter/gather DMA or uses full 32-bit addresses. Or, the driver for a subordinate device can determine the DMA channel number from the resource list that the driver receives in the IRP_MN_START_DEVICE request that starts the device. The driver uses the **DEVICE_DESCRIPTION** structure to pass this information to the I/O manager.

Before calling **IoGetDmaAdapter**, the driver should first zero-initialize the entire **DEVICE_DESCRIPTION** structure, then fill in selected members to describe the device.

The **InterfaceType** member specifies the type of bus interface that will be used for DMA. If you set **InterfaceType** to **InterfaceTypeUndefined**, **IoGetDmaAdapter** queries the PDO to determine the correct interface type for your device. Or, you can specify an explicit interface type, such as **Internal**, **Isa**, **Eisa**, or **PCIBus**. For more information, see the list of supported interface types in INTERFACE_TYPE.

If the **ScatterGather** member is set to **TRUE** and the **InterfaceType** member is set to **PCIBus**, the **Dma32BitAddresses** member is ignored and **IoGetDmaAdapter** assumes that the device supports 32-bit DMA addresses.

If the **Dma64BitAddresses** member is set to **TRUE**, the **Dma32BitAddresses** member is ignored and **IoGetDmaAdapter** assumes that the device supports 64-bit DMA addresses.

To indicate that the DMA controller hardware cannot reliably maintain an accurate transfer count, set **IgnoreCount** to **TRUE**, and set **Version** to a value other than DEVICE_DESCRIPTION_VERSION. In a platform that has such a DMA controller, the operating system ignores the DMA transfer counter but must take special

precautions to maintain data integrity during DMA operations. Typically, the use of a workaround to compensate for a deficient DMA controller degrades the speed of DMA transfers.

A driver should specify **TypeF** as the **DmaSpeed** value only if the computer's ACPI firmware supports it.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

CM_RESOURCE_LIST

DMA_ADAPTER

INTERFACE_TYPE

IRP_MN_START_DEVICE

IoGetDmaAdapter

DEVICE_INSTALL_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_INSTALL_STATE** enumeration describes a device's installation state.

Syntax

```
typedef enum _DEVICE_INSTALL_STATE {
  InstallStateInstalled,
  InstallStateNeedsReinstall,
  InstallStateFailedInstall,
  InstallStateFinishInstall
} DEVICE_INSTALL_STATE, *PDEVICE_INSTALL_STATE;
```

Constants

InstallStateInstalled	The device is installed.
InstallStateNeedsReinstall	The system will try to reinstall the device on a later enumeration.
InstallStateFailedInstall	The device did not install properly.
InstallStateFinishInstall	The installation of this device is not yet complete.

Remarks

The IoGetDeviceProperty routine supplies a **DEVICE_INSTALL_STATE** enumeration value when a driver requests **DevicePropertyInstallState**. The operating system uses the value as a hint about the install state of the device.

Requirements

Minimum supported client	Available in Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h)

See Also

DEVICE_REGISTRY_PROPERTY

IoGetDeviceProperty

DEVICE_INTERFACE_CHANGE_NOTIFICATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_INTERFACE_CHANGE_NOTIFICATION** structure describes a device interface that has been enabled (arrived) or disabled (removed). The PnP manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryDeviceInterfaceChange** events.

Syntax

```
typedef struct _DEVICE_INTERFACE_CHANGE_NOTIFICATION {
  USHORT          Version;
  USHORT          Size;
  GUID            Event;
  GUID            InterfaceClassGuid;
  PUNICODE_STRING SymbolicLinkName;
} DEVICE_INTERFACE_CHANGE_NOTIFICATION, *PDEVICE_INTERFACE_CHANGE_NOTIFICATION;
```

Members

`Version`

Specifies the version of the data structure, currently 1.

`Size`

Specifies the size of the structure, in bytes, including the size of the standard first three members plus the event-specific data.

`Event`

Specifies a GUID identifying the event: GUID_DEVICE_INTERFACE_ARRIVAL or GUID_DEVICE_INTERFACE_REMOVAL. The GUIDs are defined in Wdmguid.h.

`InterfaceClassGuid`

Specifies the class of the device interface that has just been enabled or disabled.

`SymbolicLinkName`

Pointer to a Unicode string that contains the name of the symbolic link for the device interface.

Remarks

This structure is allocated from paged memory.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

HWPROFILE_CHANGE_NOTIFICATION

IoRegisterPlugPlayNotification

PLUGPLAY_NOTIFICATION_HEADER

TARGET_DEVICE_REMOVAL_NOTIFICATION

DEVICE_OBJECT structure

1/30/2019 • 11 minutes to read • Edit Online

The **DEVICE_OBJECT** structure is used by the operating system to represent a device object. A device object represents a logical, virtual, or physical device for which a driver handles I/O requests.

Syntax

```
typedef struct _DEVICE_OBJECT {
  CSHORT                    Type;
  USHORT                    Size;
  LONG                      ReferenceCount;
  struct _DRIVER_OBJECT     *DriverObject;
  struct _DEVICE_OBJECT     *NextDevice;
  struct _DEVICE_OBJECT     *AttachedDevice;
  struct _IRP               *CurrentIrp;
  PIO_TIMER                 Timer;
  ULONG                     Flags;
  ULONG                     Characteristics;
  __volatile PVPB           Vpb;
  PVOID                     DeviceExtension;
  DEVICE_TYPE               DeviceType;
  CCHAR                     StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
  ULONG                     AlignmentRequirement;
  KDEVICE_QUEUE             DeviceQueue;
  KDPC                      Dpc;
  ULONG                     ActiveThreadCount;
  PSECURITY_DESCRIPTOR      SecurityDescriptor;
  KEVENT                    DeviceLock;
  USHORT                    SectorSize;
  USHORT                    Spare1;
  struct _DEVOBJ_EXTENSION *DeviceObjectExtension;
  PVOID                     Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

Members

`Type`

Used by the operating system to indicate that an object is a device object. For device objects, the value of this member is 3. This is a read-only member.

`Size`

Specifies the size, in bytes, of the device object. This size includes the driver-specified device extension pointed to by the **DeviceExtension** member, but does not include the opaque device object extension pointed to by the **DeviceObjectExtension** member. **Size** is a read-only member.

`ReferenceCount`

Used by the I/O manager to track the number of open handles for the device that are associated with the device object. This allows the I/O manager to avoid unloading a driver when there are outstanding handles for the driver's

device(s). This is a read-only member.

`DriverObject`

A pointer to the driver object (DRIVER_OBJECT), that represents the loaded image of the driver that was input to the DriverEntry and AddDevice routines. This member is set by the I/O manager upon a successful call to IoCreateDevice or IoCreateDeviceSecure. This is a read-only member.

`NextDevice`

A pointer to the next device object, if any, that was created by the same driver. The I/O manager updates this list at each successful call to IoCreateDevice or IoCreateDeviceSecure.

A non- Plug and Play (PnP) driver that is being unloaded must traverse ("walk") the list of its device objects and delete them. A PnP driver does not have to walk this list of device objects. Instead, PnP drivers perform their cleanup during the device removal PnP operation (IRP_MN_REMOVE_DEVICE).

A driver that recreates its device objects dynamically also uses this member. This is a read/write member.

`AttachedDevice`

A pointer to the attached device object. If there is no attached device object, this member is **NULL**. The device object that is pointed to by the **AttachedDevice** member typically is the device object of a filter driver, which intercepts I/O requests originally targeted to the device represent by the device object. For more information, see the IoAttachDevice and IoAttachDeviceByPointer topics. This is an opaque member.

`CurrentIrp`

A pointer to the current IRP if the driver has a StartIo routine whose entry point was set in the driver object and if the driver is currently processing IRP(s). Otherwise, this member is **NULL**. For more information, see the IoStartPacket and IoStartNextPacket topics. This is a read-only member.

`Timer`

A pointer to a timer object. This allows the I/O manager to call a driver-supplied timer routine every second. For more information, see IoInitializeTimer. This is a read/write member.

`Flags`

Device drivers perform a bitwise **OR** operation with this member in their newly created device objects by using one or more of the following system-defined values:

DO_BUFFERED_IO or DO_DIRECT_IO

Specifies the type of buffering that is used by the I/O manager for I/O requests that are sent to the device stack. Higher-level drivers OR this member with the same value as the next-lower driver in the stack, except possibly for highest-level drivers.

DO_BUS_ENUMERATED_DEVICE

The operating system sets this flag in each physical device object (PDO). Drivers must not modify this flag.

DO_DEVICE_INITIALIZING

The I/O manager sets this flag when it creates the device object. A device function driver or filter driver clears the flag in its AddDevice routine, after it does the following:

- Attaches the device object to the device stack.
- Establishes the device power state.
- Performs a bitwise OR operation on the member with one of the power flags (if it is necessary).

The Plug and Play (PnP) manager checks that the flag is clear after the AddDevice routine returns.

DO_EXCLUSIVE

Indicates that the driver services an exclusive device, such as a video, serial, parallel, or sound device. WDM drivers must not set this flag. For more information, see the Specifying Exclusive Access to Device Objects topic.

DO_MAP_IO_BUFFER

This flag is no longer used. Drivers should not set this flag.

DO_POWER_INRUSH

Drivers of devices that require inrush current when the device is turned on must set this flag. A driver cannot set both this flag and DO_POWER_PAGABLE.

DO_POWER_PAGABLE

Pageable drivers that are compatible with Microsoft Windows 2000 and later versions of Windows, are not part of the paging path, and do not require inrush current must set this flag. The system calls such drivers at IRQL = PASSIVE_LEVEL. Drivers cannot set both this flag and DO_POWER_INRUSH. All drivers for WDM, Microsoft Windows 98, and Windows Millennium Edition must set DO_POWER_PAGABLE.

DO_SHUTDOWN_REGISTERED

Used by the I/O manager to indicate that a driver has registered the device object for shutdown notifications. This flag should not be used by drivers.

DO_VERIFY_VOLUME

Removable-media drivers set this flag while they process transfer requests. Such drivers should also check for this flag in the target for a transfer request before they transfer any data. For more information, see the Supporting Removable Media topic.

For more information about how to set the **Flags** member, see Initializing a Device Object.

`Characteristics`

Specifies one or more system-defined constants, combined with a bitwise OR operation, that provide additional information about the driver's device. These constants include the following:

FILE_AUTOGENERATED_DEVICE_NAME

Directs the I/O manager to generate a name for the device, instead of the caller specifying a *DeviceName* when it calls this routine. The I/O manager makes sure that the name is unique. This characteristic is typically specified by a PnP bus driver to generate a name for a physical device object (PDO) for a child device on the same bus. This characteristic is new starting with Microsoft Windows 2000 and Microsoft Windows 98.

FILE_CHARACTERISTIC_PNP_DEVICE

Indicates that the device object is part of a Plug and Play (PnP) stack. This characteristic is required if a bus driver (or bus filter driver) registers WMI support for a device object that has not yet received the IRP_MN_START_DEVICE request. FILE_CHARACTERISTIC_PNP_DEVICE is also required if a function or filter driver registers for WMI before attaching to its device stack.

FILE_CHARACTERISTIC_TS_DEVICE

Indicates that the device object is part of a Terminal Services device stack. Drivers should not set this characteristic.

FILE_CHARACTERISTIC_WEBDAV_DEVICE

Indicates that a Web-based Distributed Authoring and Versioning (WebDAV) file system is mounted on the device. Drivers should not set this characteristic.

FILE_DEVICE_IS_MOUNTED

Indicates that a file system is mounted on the device. Drivers should not set this characteristic.

FILE_DEVICE_SECURE_OPEN

Directs the I/O manager to apply the security descriptor of the device object to relative opens and trailing file name opens for the device. For more information, see the Controlling Device Namespace Access topic.

FILE_FLOPPY_DISKETTE

Indicates that the device is a floppy disk device.

FILE_READ_ONLY_DEVICE

Indicates that the device cannot be written to.

FILE_REMOTE_DEVICE

Indicates that the device is remote.

FILE_REMOVABLE_MEDIA

Indicates that the storage device supports removable media. Notice that this characteristic indicates removable *media*, not a removable *device*. For example, drivers for JAZ drive devices should specify this characteristic, but drivers for PCMCIA flash disks should not.

FILE_VIRTUAL_VOLUME

Indicates that the volume is virtual. Drivers should not set this characteristic.

FILE_WRITE_ONCE_MEDIA

Indicates that the device supports write-once media. Drivers do not set this member directly. For more information about how to set device characteristics, see the Specifying Device Characteristics topic.

FILE_CHARACTERISTIC_CSV

Indicates that the device is a Cluster Shared Volume (CSV).

FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL

The IO Manager normally performs a full security check for traverse access on every file open when the client is an app container. Setting of this flag bypasses this enforced traverse access check if the client token already has traverse privileges.

FILE_PORTABLE_DEVICE

Indicates that the underlying stack considers the device portable. This is used by the storage stack and means that the device is not in the local machine container and is not on a fixed bus type.

`Vpb`

A pointer to the volume parameter block (VPB) that is associated with the device object. For file system drivers, the VPB can provide a connection to any unnamed logical device object that represents an instance of a mounted volume. This is an opaque member.

`DeviceExtension`

A pointer to the device extension. The structure and contents of the device extension are driver-defined. The size is driver-determined, specified in the driver's call to IoCreateDevice or IoCreateDeviceSecure. For more information about device extensions, see Device Extensions. This is a read-only member. However, the object that the member points to can be modified by the driver.

`DeviceType`

Set by IoCreateDevice and IoCreateDeviceSecure by using the value that is specified for that routine's *DeviceType* parameter. For more information, see the Specifying Device Types topic.

`StackSize`

Specifies the minimum number of stack locations in IRPs to be sent to this driver. IoCreateDevice and IoCreateDeviceSecure set this member to 1 in newly created device objects; lowest-level drivers can therefore ignore this member. The I/O manager automatically sets the **StackSize** member in a higher-level driver's device object to the appropriate value if the driver calls IoAttachDevice or IoAttachDeviceToDeviceStack. Only a higher-level driver that chains itself over another driver with **IoGetDeviceObjectPointer** must explicitly set the value of **StackSize** in its own device object(s) to 1 + the **StackSize** value of the next-lower driver's device object.

`Queue`

Used internally by the I/O manager to queue the device object when it is required. This is an opaque member.

`Queue.ListEntry`

A LIST_ENTRY structure that contains forward and backward pointers for a doubly linked list.

`Queue.Wcb`

Device context information used by I/O manager.

`AlignmentRequirement`

Specifies the device's address alignment requirement for data transfers. The value must be one of the FILE_*XXX*_ALIGNMENT values that are defined in Wdm.h. For more information, see the Initializing a Device Object, GetDmaAlignment, and ZwQueryInformationFile topics.

`DeviceQueue`

The device queue object for the device object. The device queue object contains any IRPs that are waiting to be processed by the driver that is associated with the device object. For more information, see the Driver-Managed IRP Queues topic. This is an opaque member.

`Dpc`

The deferred procedure call (DPC) object for the device object. For more information, see the Introduction to DPC Objects topic. This is an opaque member.

`ActiveThreadCount`

Reserved for future use. This is an opaque member.

`SecurityDescriptor`

Specifies a security descriptor (SECURITY_DESCRIPTOR) for the device object when the device object is created. If this member is **NULL**, the device object receives default security settings. This is a read-only member, although the member can be modified through the ZwSetSecurityObjectfunction.

`DeviceLock`

A synchronization event object that is allocated by the I/O manager. The I/O manager obtains his event object before it dispatches a mount or mount-verify request to a file-system driver. This is an opaque member.

`SectorSize`

If the device object does not represent a volume, this member is set to zero. If the device object represents a volume, this member specifies the volume's sector size, in bytes. The I/O manager uses this member to make sure that all read operations, write operations, and set file position operations that are issued are aligned correctly when intermediate buffering is disabled. A default system bytes-per-sector value is used when the device object is created, however, file system drivers; and more rarely, legacy and minifilter drivers, can update this value that is based on the geometry of the underlying volume hardware when a mount occurs. Other drivers should not modify this member.

`Spare1`

Reserved for system use. This is an opaque member.

`DeviceObjectExtension`

A pointer to a device object extension that is used by the I/O manager and PnP manager to store information about the state of the device. This is an opaque member.

`Reserved`

Reserved for system use. This is an opaque member.

Remarks

The operating system represents devices by device objects. For more information, see the Device Objects and Device Stacks topic.

Drivers create device objects by using the IoCreateDevice and IoCreateDeviceSecure routines. For more information about how to create device objects, see Creating a Device Object.

A device object is partially opaque. Drivers do not set members of the device object directly, unless otherwise documented. For more information about the members that drivers can modify directly, see Initializing a Device Object. For information about other device object properties, see Properties of Device Objects.

Opaque members within a device object must be considered inaccessible. Drivers that have dependencies on object member locations or access to opaque members might not remain portable and interoperable with other drivers over time.

The system-supplied video port driver sets up the fields of the device objects that it creates on behalf of video miniport drivers.

The system-supplied SCSI port driver sets up the fields of the device objects that it creates on behalf of SCSI miniport drivers.

The system-supplied NDIS library sets up the fields of the device objects that it creates on behalf of NDIS miniport drivers.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

DRIVER_OBJECT

IoAttachDevice

IoAttachDeviceToDeviceStack

IoCreateDevice

IoDeleteDevice

IoGetDeviceObjectPointer

DEVICE_POWER_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_POWER_STATE** enumeration type indicates a device power state.

Syntax

```
typedef enum _DEVICE_POWER_STATE {
  PowerDeviceUnspecified,
  PowerDeviceD0,
  PowerDeviceD1,
  PowerDeviceD2,
  PowerDeviceD3,
  PowerDeviceMaximum
} DEVICE_POWER_STATE, *PDEVICE_POWER_STATE;
```

Constants

PowerDeviceUnspecified	Indicates an unspecified device power state.
PowerDeviceD0	Indicates a maximum device power state, which corresponds to device working state D0.
PowerDeviceD1	Indicates a device sleeping state less than **PowerDeviceD0** and greater than **PowerDeviceD2**, which corresponds to device power state D1.
PowerDeviceD2	Indicates a device sleeping state less than **PowerDeviceD1** and greater than **PowerDeviceD3**, which corresponds to device power state D2.
PowerDeviceD3	Indicates the lowest-powered device sleeping state, which corresponds to device power state D3.
PowerDeviceMaximum	The number of device power state values for this enumeration type that represent actual power states. The value of the other device power states is less than this value.

Remarks

A power state indicates the level of power consumption—and thus the extent of computing activity—by the system or by a single device. The power manager sets the power state of the system as a whole, where the system power state is indicated by one of the values of the SYSTEM_POWER_STATE enumeration type. Device drivers set the power state of their individual devices, where the device power state is indicated by one of the values of the **DEVICE_POWER_STATE** enumeration type.

For more information about system power states, see Handling System Power State Requests. For more information about device power states, see Managing Power for Individual Devices. For more information about power management in general, see Introduction to Power Management.

Requirements

Header	wdm.h (include Wdm.h, Ntpoapi.h)

See Also

SYSTEM_POWER_STATE

DEVICE_REMOVAL_POLICY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_REMOVAL_POLICY** enumeration describes a device's removal policy.

Syntax

```
typedef enum _DEVICE_REMOVAL_POLICY {
  RemovalPolicyExpectNoRemoval,
  RemovalPolicyExpectOrderlyRemoval,
  RemovalPolicyExpectSurpriseRemoval
} DEVICE_REMOVAL_POLICY, *PDEVICE_REMOVAL_POLICY;
```

Constants

RemovalPolicyExpectNoRemoval	The device is not typically removed.
RemovalPolicyExpectOrderlyRemoval	The device is typically removed in an orderly fashion. (Before the device is removed, the Plug and Play [PnP] manager sends an IRP_MN_QUERY_REMOVE_DEVICE request to the device's driver.)
RemovalPolicyExpectSurpriseRemoval	The device can be removed suddenly. (The driver receives no advance warning that the device will be removed. The Plug and Play [PnP] manager sends an IRP_MN_SURPRISE_REMOVAL request when the device is removed.)

Remarks

The IoGetDeviceProperty routine supplies a **DEVICE_REMOVAL_POLICY** enumeration value when a driver requests **DevicePropertyRemovalPolicy**. The operating system uses the value as a hint as to how the device is typically removed from the computer.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h)

See Also

IRP_MN_QUERY_REMOVE_DEVICE

IRP_MN_SURPRISE_REMOVAL

IoGetDeviceProperty

DEVICE_RESET_INTERFACE_STANDARD structure

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_RESET_INTERFACE_STANDARD** structure enables function drivers to reset and recover malfunctioning devices. This structure describes the GUID_DEVICE_RESET_INTERFACE_STANDARD interface. For more information see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD.

Syntax

```
typedef struct _DEVICE_RESET_INTERFACE_STANDARD {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
  PDEVICE_RESET_HANDLER  DeviceReset;
  ULONG                  SupportedResetTypes;
  PVOID                  Reserved;
} DEVICE_RESET_INTERFACE_STANDARD, *PDEVICE_RESET_INTERFACE_STANDARD;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined interface version.

`Context`

A pointer to interface-specific context information

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the interface's reference count.

`DeviceReset`

A pointer to the interface's DeviceReset routine. This routine can be used by function drivers to attempt to reset and recover a malfunctioning device.

`SupportedResetTypes`

`Reserved`

This member is reserved for future use.

Remarks

The **DEVICE_RESET_INTERFACE_STANDARD** structure is an extension of the INTERFACE structure. A driver

obtains a pointer to the **DEVICE_RESET_INTERFACE_STANDARD** structure by sending an IRP_MN_QUERY_INTERFACE IRP to its bus driver with **InterfaceType** set to GUID_DEVICE_RESET_INTERFACE_STANDARD.

For more information about the purpose of this interface, see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h

See Also

DeviceReset

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

DEVICE_RESET_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_RESET_TYPE** enumeration specifies the type of device reset that is being requested by a call to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface. For more information see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD.

Syntax

```
typedef enum _DEVICE_RESET_TYPE {
  FunctionLevelDeviceReset,
  PlatformLevelDeviceReset
} DEVICE_RESET_TYPE;
```

Constants

FunctionLevelDeviceReset	A function-level device reset, which is restricted to a specific device.
PlatformLevelDeviceReset	A platform-level device reset, which affects a specific device and all other devices that are connected to it via the same power rail or reset line.

Requirements

Header	wdm.h

See Also

DEVICE_RESET_INTERFACE_STANDARD

DeviceReset

Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD

DEVICE_WAKE_DEPTH Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **DEVICE_WAKE_DEPTH** enumeration specifies the deepest device power state from which a device can trigger a wake signal.

Syntax

```
typedef enum _DEVICE_WAKE_DEPTH {
  DeviceWakeDepthNotWakeable,
  DeviceWakeDepthD0,
  DeviceWakeDepthD1,
  DeviceWakeDepthD2,
  DeviceWakeDepthD3hot,
  DeviceWakeDepthD3cold,
  DeviceWakeDepthMaximum
} DEVICE_WAKE_DEPTH, *PDEVICE_WAKE_DEPTH;
```

Constants

DeviceWakeDepthNotWakeable	There is no device power state that can trigger a wake signal.
DeviceWakeDepthD0	D0 is the deepest device power state from which the device can trigger a wake signal. For more information, see Remarks.
DeviceWakeDepthD1	D1 is the deepest low-power device power state from which the device can trigger a wake signal.
DeviceWakeDepthD2	D2 is the deepest low-power device power state from which the device can trigger a wake signal.
DeviceWakeDepthD3hot	D3hot is the deepest low-power device power state from which the device can trigger a wake signal.
DeviceWakeDepthD3cold	D3cold is the deepest low-power device power state from which the device can trigger a wake signal.
DeviceWakeDepthMaximum	Reserved for use by the operating system.

Remarks

The *DeepestWakeableDstate* parameter of the GetIdleWakeInfo routine is a pointer to a variable of type **DEVICE_WAKE_DEPTH**.

The drivers for most devices have no reason to arm a wake signal when the device is in D0. These drivers can treat the **DeviceWakeDepthD0** output value as equivalent to a call to the *GetIdleWakeInfo* routine that fails and returns an error status.

Requirements

Header	wdm.h (include Wdm.h)

See Also

GetIdleWakeInfo

DMA_ADAPTER structure

1/30/2019 • 2 minutes to read • Edit Online

The **DMA_ADAPTER** structure describes a system-defined interface to a DMA controller for a given device. A driver calls IoGetDmaAdapter to obtain this structure.

Syntax

```
typedef struct _DMA_ADAPTER {
  USHORT          Version;
  USHORT          Size;
  PDMA_OPERATIONS DmaOperations;
} *PADAPTER_OBJECT, DMA_ADAPTER, *PDMA_ADAPTER;
```

Members

`Version`

Specifies the version of this structure. Version 3 of the **DMA_ADAPTER** structure is available starting with Windows 8. For versions 1 and 2 of this structure, this member is set to the value 1. For more information, see the following Remarks section.

`Size`

Specifies the size, in bytes, of this structure.

`DmaOperations`

Pointer to a DMA_OPERATIONS structure that contains pointers to DMA adapter functions. The version of the **DMA_OPERATIONS** structure that this member points to is determined by the version of the **DMA_ADAPTER** structure. Thus, for version 1 of the **DMA_ADAPTER** structure, *DmaOperations* points to version 1 of the **DMA_OPERATIONS** structure, and so on. For more information about structure versions, see the following Remarks section.

Remarks

Drivers for devices that use DMA to transfer data use this structure to obtain the addresses of functions that enable use of a DMA controller. Usually, drivers obtain this structure by calling the IoGetDmaAdapter routine. Drivers can also obtain this structure by querying for the BUS_INTERFACE_STANDARD interface.

> **Note** When the **IoGetDmaAdapter** routine returns a pointer to version 1 or version 2 of the **DMA_ADAPTER** structure, this routine always sets the **Version** member of this structure to 1. Thus, the caller cannot use the **Version** member to distinguish between versions 1 and 2 of the **DMA_ADAPTER** structure. For more information about how to determine the version of a **DMA_ADAPTER** structure that is returned by this routine, see IoGetDmaAdapter.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BUS_INTERFACE_STANDARD

DMA_OPERATIONS

IoGetDmaAdapter

DMA_ADAPTER_INFO structure

2/5/2019 • 2 minutes to read • Edit Online

The **DMA_ADAPTER_INFO** structure is a container for a **DMA_ADAPTER_INFO**_*XXX* structure that describes the capabilities of a system DMA controller.

Syntax

```
typedef struct _DMA_ADAPTER_INFO {
  ULONG Version;
  union {
    DMA_ADAPTER_INFO_V1 V1;
  };
} DMA_ADAPTER_INFO, *PDMA_ADAPTER_INFO;
```

Members

`Version`

The version number of the **DMA_ADAPTER_INFO**_*XXX* structure that follows this member. For a DMA_ADAPTER_INFO_V1 structure, set this member to DMA_ADAPTER_INFO_VERSION1 before calling the GetDmaAdapterInfo routine.

`V1`

The capabilities of the bus-master DMA device or the system DMA controller. For more information, see DMA_ADAPTER_INFO_V1.

Remarks

A device driver calls the GetDmaAdapterInfo routine to obtain a **DMA_ADAPTER_INFO** structure that describes the capabilities of a DMA controller.

The unnamed union in this structure contains a **DMA_ADAPTER_INFO**_*XXX* structure. The **Version** member indicates which version of the **DMA_ADAPTER_INFO**_*XXX* structure is contained in the union. Currently, only version 1 is supported.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DMA_ADAPTER_INFO_V1

GetDmaAdapterInfo

DMA_ADAPTER_INFO_V1 structure

1/30/2019 • 2 minutes to read • Edit Online

The **DMA_ADAPTER_INFO_V1** structure describes the capabilities of the system DMA controller that is represented by an adapter object.

Syntax

```
typedef struct _DMA_ADAPTER_INFO_V1 {
  ULONG ReadDmaCounterAvailable;
  ULONG ScatterGatherLimit;
  ULONG DmaAddressWidth;
  ULONG Flags;
  ULONG MinimumTransferUnit;
} DMA_ADAPTER_INFO_V1, *PDMA_ADAPTER_INFO_V1;
```

Members

`ReadDmaCounterAvailable`

Whether the counter value in each DMA channel can be read. This member is **TRUE** if the counter can be read, and is **FALSE** if it cannot be read.

`ScatterGatherLimit`

The maximum number of elements in a scatter/gather list that the DMA controller can process in a single scatter/gather DMA transfer.

`DmaAddressWidth`

The memory address width, in bits, of the DMA controller. The width is expressed as the number of bits in a DMA address. If the DMA address width is less than the memory address width, the platform hardware drives the remaining, high-order memory address bits to zero during a DMA transfer.

`Flags`

A set of flags that describe the capabilities of the DMA adapter. No flags are currently defined for this member.

`MinimumTransferUnit`

The size, in bytes, of the minimum transfer unit. The number of bytes specified by an element in a scatter/gather list must be an integer multiple of the minimum transfer unit.

Remarks

The **V1** member of the DMA_ADAPTER_INFO structure is a structure of type **DMA_ADAPTER_INFO_V1**.

A driver calls GetDmaAdapterInfo to obtain information about the hardware capabilities of a system DMA channel. **GetDmaAdapterInfo** writes this information into the **V1** member of a caller-supplied **DMA_ADAPTER_INFO** structure.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DMA_ADAPTER_INFO

GetDmaAdapterInfo

DMA_OPERATIONS structure

1/30/2019 • 6 minutes to read • Edit Online

The **DMA_OPERATIONS** structure provides a table of pointers to functions that control the operation of a DMA controller.

Syntax

```
typedef struct _DMA_OPERATIONS {
  ULONG                                 Size;
  PPUT_DMA_ADAPTER                      PutDmaAdapter;
  PALLOCATE_COMMON_BUFFER               AllocateCommonBuffer;
  PFREE_COMMON_BUFFER                   FreeCommonBuffer;
  PALLOCATE_ADAPTER_CHANNEL             AllocateAdapterChannel;
  PFLUSH_ADAPTER_BUFFERS                FlushAdapterBuffers;
  PFREE_ADAPTER_CHANNEL                 FreeAdapterChannel;
  PFREE_MAP_REGISTERS                   FreeMapRegisters;
  PMAP_TRANSFER                         MapTransfer;
  PGET_DMA_ALIGNMENT                    GetDmaAlignment;
  PREAD_DMA_COUNTER                     ReadDmaCounter;
  PGET_SCATTER_GATHER_LIST              GetScatterGatherList;
  PPUT_SCATTER_GATHER_LIST              PutScatterGatherList;
  PCALCULATE_SCATTER_GATHER_LIST_SIZE CalculateScatterGatherList;
  PBUILD_SCATTER_GATHER_LIST            BuildScatterGatherList;
  PBUILD_MDL_FROM_SCATTER_GATHER_LIST BuildMdlFromScatterGatherList;
  PGET_DMA_ADAPTER_INFO                 GetDmaAdapterInfo;
  PGET_DMA_TRANSFER_INFO                GetDmaTransferInfo;
  PINITIALIZE_DMA_TRANSFER_CONTEXT    InitializeDmaTransferContext;
  PALLOCATE_COMMON_BUFFER_EX            AllocateCommonBufferEx;
  PALLOCATE_ADAPTER_CHANNEL_EX          AllocateAdapterChannelEx;
  PCONFIGURE_ADAPTER_CHANNEL            ConfigureAdapterChannel;
  PCANCEL_ADAPTER_CHANNEL               CancelAdapterChannel;
  PMAP_TRANSFER_EX                      MapTransferEx;
  PGET_SCATTER_GATHER_LIST_EX           GetScatterGatherListEx;
  PBUILD_SCATTER_GATHER_LIST_EX         BuildScatterGatherListEx;
  PFLUSH_ADAPTER_BUFFERS_EX             FlushAdapterBuffersEx;
  PFREE_ADAPTER_OBJECT                  FreeAdapterObject;
  PCANCEL_MAPPED_TRANSFER               CancelMappedTransfer;
  PALLOCATE_DOMAIN_COMMON_BUFFER        AllocateDomainCommonBuffer;
  PFLUSH_DMA_BUFFER                     FlushDmaBuffer;
  PJOIN_DMA_DOMAIN                      JoinDmaDomain;
  PLEAVE_DMA_DOMAIN                     LeaveDmaDomain;
  PGET_DMA_DOMAIN                       GetDmaDomain;
  PALLOCATE_COMMON_BUFFER_WITH_BOUNDS AllocateCommonBufferWithBounds;
} *PDMA_OPERATIONS, DMA_OPERATIONS;
```

Members

`Size`

The size, in bytes, of this **DMA_OPERATIONS** structure.

`PutDmaAdapter`

A pointer to a system-defined routine to free a DMA_ADAPTER structure. For more information, see PutDmaAdapter.

`AllocateCommonBuffer`

A pointer to a system-defined routine to allocate a physically contiguous DMA buffer. For more information, see AllocateCommonBuffer.

`FreeCommonBuffer`

A pointer to a system-defined routine to free a physically contiguous DMA buffer previously allocated by **AllocateCommonBuffer**. For more information, see FreeCommonBuffer.

`AllocateAdapterChannel`

A pointer to a system-defined routine to allocate a channel for DMA operations. For more information, see AllocateAdapterChannel.

`FlushAdapterBuffers`

A pointer to a system-defined routine to flush data from the system or bus-master adapter's internal cache after a DMA operation. For more information, see FlushAdapterBuffers.

`FreeAdapterChannel`

A pointer to a system-defined routine to free a channel previously allocated for DMA operations by **AllocateAdapterChannel**. For more information, see FreeAdapterChannel.

`FreeMapRegisters`

A pointer to a system-defined routine to free map registers allocated for DMA operations. For more information, see FreeMapRegisters.

`MapTransfer`

A pointer to a system-defined routine to begin a DMA operation. For more information, see MapTransfer.

`GetDmaAlignment`

A pointer to a system-defined routine to obtain the DMA alignment requirements of the controller. For more information, see GetDmaAlignment.

`ReadDmaCounter`

A pointer to a system-defined routine to obtain the current transfer count for a DMA operation. For more information, see ReadDmaCounter.

`GetScatterGatherList`

A pointer to a system-defined routine that allocates map registers and creates a scatter/gather list for DMA. For more information, see GetScatterGatherList.

`PutScatterGatherList`

A pointer to a system-defined routine that frees map registers and a scatter/gather list after a DMA operation is complete. For more information, see PutScatterGatherList.

`CalculateScatterGatherList`

A pointer to a system-defined routine that determines the buffer size needed to hold the scatter/gather list that describes an I/O data buffer. This member is available only in versions 2 and later of **DMA_OPERATIONS**. For more information, see CalculateScatterGatherList.

`BuildScatterGatherList`

A pointer to a system-defined routine that allocates map registers and creates a scatter/gather list for DMA in a driver-supplied buffer. This member is available only in versions 2 and later of **DMA_OPERATIONS**. For more

information, see BuildScatterGatherList.

`BuildMdlFromScatterGatherList`

A pointer to a system-defined routine that builds an MDL corresponding to a scatter/gather list. This member is available only in versions 2 and later of **DMA_OPERATIONS**. For more information, see BuildMdlFromScatterGatherList.

`GetDmaAdapterInfo`

A pointer to a system-defined routine that describes the capabilities of a bus-master DMA device or a system DMA controller. **GetDmaAdapterInfo** is available only in version 3 of **DMA_OPERATIONS**. For more information, see GetDmaAdapterInfo.

`GetDmaTransferInfo`

A pointer to a system-defined routine that describes the allocation requirements for a scatter/gather list. This routine replaces CalculateScatterGatherList. **GetDmaTransferInfo** is available only in version 3 of **DMA_OPERATIONS**. For more information, see GetDmaTransferInfo.

`InitializeDmaTransferContext`

A pointer to a system-defined routine that initializes an opaque DMA transfer context. The operating system stores the internal status of a DMA transfer in this context. **InitializeDmaTransferContext** is available only in version 3 of **DMA_OPERATIONS**. For more information, see InitializeDmaTransferContext.

`AllocateCommonBufferEx`

A pointer to a system-defined routine that allocates memory for a common buffer and maps this memory so that it can accessed both by the processor and by a DMA device. **AllocateCommonBufferEx** is available only in version 3 of **DMA_OPERATIONS**. For more information, see AllocateCommonBufferEx.

`AllocateAdapterChannelEx`

A pointer to a system-defined routine that allocates the resources required for a DMA transfer and then calls the driver-supplied *AdapterControl* routine to initiate the DMA transfer. **AllocateAdapterChannelEx** is available only in version 3 of **DMA_OPERATIONS**. For more information, see AllocateAdapterChannelEx.

`ConfigureAdapterChannel`

A pointer to a system-defined routine enables a custom function that is implemented by the DMA controller. **ConfigureAdapterChannel** is available only in version 3 of **DMA_OPERATIONS**. For more information, see ConfigureAdapterChannel.

`CancelAdapterChannel`

A pointer to a system-defined routine that tries to cancel a pending request to allocate a DMA channel. **CancelAdapterChannel** is available only in version 3 of **DMA_OPERATIONS**. For more information, see CancelAdapterChannel.

`MapTransferEx`

A pointer to a system-defined routine that sets up map registers to map the physical addresses in a scatter/gather list to the logical addresses that are required to do a DMA transfer. **MapTransferEx** is available only in version 3 of **DMA_OPERATIONS**. For more information, see MapTransferEx.

`GetScatterGatherListEx`

A pointer to a system-defined routine that allocates resources required for a DMA transfer, builds a scatter/gather list, and then calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.

GetScatterGatherListEx is available only in version 3 of **DMA_OPERATIONS**. For more information, see GetScatterGatherListEx. This routine is a wrapper of **AllocateAdapterChannelEx** and **MapTransferEx**.

`BuildScatterGatherListEx`

A pointer to a system-defined routine that builds a scatter/gather list in a caller-allocated buffer, and then calls the driver-supplied *AdapterListControl* routine to initiate the DMA transfer. **BuildScatterGatherListEx** is available only in version 3 of **DMA_OPERATIONS**. For more information, see BuildScatterGatherListEx.

`FlushAdapterBuffersEx`

A pointer to a system-defined routine that flushes any data that remains in the system DMA controller's internal cache or in a bus-master adapter's internal cache at the end of a DMA transfer. For a device that uses a system DMA controller, this routine cancels the current DMA transfer on the controller if the transfer is not complete. **FlushAdapterBuffersEx** is available only in version 3 of **DMA_OPERATIONS**. For more information, see FlushAdapterBuffersEx.

`FreeAdapterObject`

A pointer to a system-defined routine that releases the specified adapter object after a driver has completed all DMA operations. **FreeAdapterObject** is available only in version 3 of **DMA_OPERATIONS**. For more information, see FreeAdapterObject.

`CancelMappedTransfer`

A pointer to a system-defined routine that cancels a mapped transfer. **CancelMappedTransfer** is available only in version 3 of **DMA_OPERATIONS**. For more information, see CancelMappedTransfer.

`AllocateDomainCommonBuffer`

A pointer to a *PALLOCATE_DOMAIN_COMMON_BUFFER* callback routine to allocate a domain common buffer. This option is available in Windows 10 and later versions.

`FlushDmaBuffer`

A pointer to a *PFLUSH_DMA_BUFFER* callback function that flushes any data remaining in the cache. This option is available in Windows 10 and later versions.

`JoinDmaDomain`

A pointer to a *PJOIN_DMA_DOMAIN* callback function that joins the specified DMA domain. This option is available in Windows 10 and later versions.

`LeaveDmaDomain`

A pointer to a *PLEAVE_DMA_DOMAIN* callback function that leaves the specified DMA domain. This option is available in Windows 10 and later versions.

`GetDmaDomain`

A pointer to the *PGET_DMA_DOMAIN* callback function that gets a handle to the DMA domain. This option is available in Windows 10 and later versions.

`AllocateCommonBufferWithBounds`

A pointer to a *PALLOCATE_COMMON_BUFFER_WITH_BOUNDS* callback function that allocates the memory for a common buffer and maps it so that it can be accessed by a master device and the CPU. The common buffer can be bound by an optional minimum and maximum logical address. This option is available starting in Windows 10, version 1803.

Remarks

All members of this structure, with the exception of **Size**, are pointers to functions that drivers use to perform DMA operations for their devices. Drivers obtain these pointers by calling the IoGetDmaAdapter routine. The version of the **DMA_OPERATIONS** structure that this routine returns depends on the **Version** member of the DEVICE_DESCRIPTION structure that is passed to **IoGetDmaAdapter** as an input parameter. If **Version** is DEVICE_DESCRIPTION_VERSION or DEVICE_DESCRIPTION_VERSION1, version 1 of this structure is returned. If **Version** is DEVICE_DESCRIPTION_VERSION2, version 2 of this structure is returned. Version 2 of **DMA_OPERATIONS** is available starting with Windows XP. If **Version** is DEVICE_DESCRIPTION_VERSION3, version 3 of this structure is returned. Version 3 of **DMA_OPERATIONS** is available starting with Windows 8.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

AllocateAdapterChannel

AllocateAdapterChannelEx

AllocateCommonBuffer

AllocateCommonBufferEx

BuildMdlFromScatterGatherList

BuildScatterGatherList

BuildScatterGatherListEx

CalculateScatterGatherList

CancelAdapterChannel

CancelMappedTransfer

ConfigureAdapterChannel

DEVICE_DESCRIPTION

FlushAdapterBuffers

FlushAdapterBuffersEx

FreeAdapterChannel

FreeAdapterObject

FreeCommonBuffer

FreeMapRegisters

GetDmaAdapterInfo

GetDmaAlignment

GetDmaTransferInfo

GetScatterGatherList

GetScatterGatherListEx

InitializeDmaTransferContext

IoGetDmaAdapter

MapTransfer

MapTransferEx

PutDmaAdapter

PutScatterGatherList

ReadDmaCounter

DMA_TRANSFER_INFO structure

2/5/2019 • 2 minutes to read • Edit Online

The **DMA_TRANSFER_INFO** structure is a container for a **DMA_TRANSFER_INFO_*XXX*** structure that describes the allocation requirements for a scatter/gather list.

Syntax

```
typedef struct _DMA_TRANSFER_INFO {
  ULONG Version;
  union {
    DMA_TRANSFER_INFO_V1 V1;
    DMA_TRANSFER_INFO_V2 V2;
  };
} DMA_TRANSFER_INFO, *PDMA_TRANSFER_INFO;
```

Members

`Version`

The version number of the **DMA_TRANSFER_INFO_*XXX*** structure that follows this member. For a DMA_TRANSFER_INFO_V1 structure, set this member to DMA_TRANSFER_INFO_VERSION1 before calling the GetDmaTransferInfo routine.

`V1`

The allocation requirements for a scatter/gather list. For more information, see DMA_TRANSFER_INFO_V1.

`V2`

The allocation requirements for a scatter/gather list. For more information, see **DMA_TRANSFER_INFO_V2**.

Remarks

A device driver calls the GetDmaTransferInfo routine to obtain a **DMA_TRANSFER_INFO** structure that describes the allocation requirements for the scatter/gather list to use in a DMA transfer.

The unnamed union in this structure contains a **DMA_TRANSFER_INFO_*XXX*** structure. The **Version** member indicates which version of the **DMA_TRANSFER_INFO_*XXX*** structure is contained in the union.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DMA_TRANSFER_INFO_V1

DMA_TRANSFER_INFO_V1 structure

1/30/2019 • 2 minutes to read • Edit Online

The **DMA_TRANSFER_INFO_V1** structure contains the allocation requirements for a scatter/gather list that describes the I/O data buffer for a DMA transfer.

Syntax

```
typedef struct _DMA_TRANSFER_INFO_V1 {
  ULONG MapRegisterCount;
  ULONG ScatterGatherElementCount;
  ULONG ScatterGatherListSize;
} DMA_TRANSFER_INFO_V1, *PDMA_TRANSFER_INFO_V1;
```

Members

`MapRegisterCount`

The number of map registers required to translate all the physical addresses in the scatter/gather list to logical addresses.

`ScatterGatherElementCount`

The number of scatter/gather elements in the scatter/gather list. Each element is a structure of type **SCATTER_GATHER_ELEMENT**.

`ScatterGatherListSize`

The required size, in bytes, of the scatter/gather buffer. This buffer contains the scatter/gather list that describes the memory that is used to buffer I/O data during the DMA transfer. The scatter/gather buffer must be large enough to contain a SCATTER_GATHER_LIST structure and an array of **SCATTER_GATHER_ELEMENT** structures, plus additional data that is used internally by the operating system.

Remarks

The **V1** member of the DMA_TRANSFER_INFO structure is a structure of type **DMA_TRANSFER_INFO_V1**.

The *Mdl*, *Offset*, and *Length* parameters to the GetDmaTransferInfo routine together describe the I/O data buffer for a DMA transfer. For the purpose of programming a DMA controller, this buffer can more conveniently be described by a scatter/gather list. (A scatter/gather list describes the memory in an I/O data buffer but does not contain this memory.) A driver calls **GetDmaTransferInfo** to obtain the allocation requirements for this scatter/gather list. **GetDmaTransferInfo** writes the allocation requirements into the **V1** member of a caller-supplied **DMA_TRANSFER_INFO** structure.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DMA_TRANSFER_INFO

GetDmaTransferInfo

SCATTER_GATHER_LIST

DRIVER_OBJECT structure

1/30/2019 • 4 minutes to read • Edit Online

Each driver object represents the image of a loaded kernel-mode driver. A pointer to the driver object is an input parameter to a driver's DriverEntry, AddDevice, and optional Reinitialize routines and to its Unload routine, if any.

A driver object is partially opaque. Driver writers must know about certain members of a driver object to initialize a driver and to unload it if the driver is unloadable. The following members of the driver object are accessible to drivers.

Syntax

```
typedef struct _DRIVER_OBJECT {
  CSHORT             Type;
  CSHORT             Size;
  PDEVICE_OBJECT     DeviceObject;
  ULONG              Flags;
  PVOID              DriverStart;
  ULONG              DriverSize;
  PVOID              DriverSection;
  PDRIVER_EXTENSION  DriverExtension;
  UNICODE_STRING     DriverName;
  PUNICODE_STRING    HardwareDatabase;
  PFAST_IO_DISPATCH  FastIoDispatch;
  PDRIVER_INITIALIZE DriverInit;
  PDRIVER_STARTIO    DriverStartIo;
  PDRIVER_UNLOAD     DriverUnload;
  PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Members

`Type`

`Size`

`DeviceObject`

Pointer to the device objects created by the driver. This member is automatically updated when the driver calls IoCreateDevice successfully. A driver can use this member and the **NextDevice** member of DEVICE_OBJECT to step through a list of all the device objects that the driver created.

`Flags`

`DriverStart`

`DriverSize`

`DriverSection`

`DriverExtension`

Pointer to the driver extension. The only accessible member of the driver extension is **DriverExtension->AddDevice**, into which a driver's **DriverEntry** routine stores the driver's AddDevice routine.

`DriverName`

`HardwareDatabase`

Pointer to the **\Registry\Machine\Hardware** path to the hardware configuration information in the registry.

`FastIoDispatch`

Pointer to a structure defining the driver's fast I/O entry points. This member is used only by FSDs and network transport drivers.

`DriverInit`

The entry point for the DriverEntry routine, which is set up by the I/O manager.

`DriverStartIo`

The entry point for the driver's StartIo routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no *StartIo* routine, this member is **NULL**.

`DriverUnload`

The entry point for the driver's Unload routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no *Unload* routine, this member is **NULL**.

`MajorFunction`

A dispatch table consisting of an array of entry points for the driver's *DispatchXxx* routines. The array's index values are the **IRP_MJ_*XXX*** values representing each IRP major function code. Each driver must set entry points in this array for the **IRP_MJ_*XXX*** requests that the driver handles. For more information, see Writing Dispatch Routines.

To help Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools, each *DispatchXxx* routine is declared using the DRIVER_DISPATCH type, as shown in this code example:

```
DRIVER_DISPATCH DispatchXxx;
```

Then, the callback routine is implemented as follows:

```
_Use_decl_annotations_
NTSTATUS
  DispatchXxx(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp
    )
  {
      // Function body
  }
```

The DRIVER_DISPATCH function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_DISPATCH function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Remarks

Each kernel-mode driver's initialization routine should be named DriverEntry so the system will load the driver automatically. If this routine's name is something else, the driver writer must define the name of the initialization routine for the linker; otherwise, the system loader or I/O manager cannot find the driver's transfer address. The names of other standard driver routines can be chosen at the discretion of the driver writer.

A driver must set its *DispatchXxx* entry points in the driver object that is passed in to the **DriverEntry** routine when the driver is loaded. A device driver must set one or more *DispatchXxx* entry points for the **IRP_MJ_*XXX*** that any driver of the same type of device is required to handle. A higher-level driver must set one or more *DispatchXxx* entry points for all the **IRP_MJ_*XXX*** that it must pass on to the underlying device driver. Otherwise, a driver is not sent IRPs for any **IRP_MJ_*XXX*** for which it does not set up a *DispatchXxx* routine in the driver object. For more information about the set of **IRP_MJ_*XXX*** that drivers for different types of underlying devices are required to handle, see IRP Major Function Codes.

The **DriverEntry** routine also sets the driver's *AddDevice*, *StartIo* and/or Unload entry points, if any, in the driver object.

The **HardwareDatabase** string can be used by device drivers to get hardware configuration information from the registry when the driver is loaded. A driver is given read-only access to this string.

The *RegistryPath* input to the **DriverEntry** routine points to the **\Registry\Machine\System\CurrentControlSet\Services<i>DriverName** key, where the value entry of *DriverName* identifies the driver. As for the **HardwareDatabase** in the input driver object, a driver is given read-only access to this string.

Undocumented members within a driver object should be considered inaccessible. Drivers with dependencies on object member locations or on access to undocumented members might not remain portable and interoperable with other drivers over time.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DriverEntry

IoCreateDevice

IoDeleteDevice

StartIo

Unload

ENLISTMENT_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **ENLISTMENT_BASIC_INFORMATION** structure contains information about an enlistment object.

Syntax

```
typedef struct _ENLISTMENT_BASIC_INFORMATION {
  GUID EnlistmentId;
  GUID TransactionId;
  GUID ResourceManagerId;
} ENLISTMENT_BASIC_INFORMATION, *PENLISTMENT_BASIC_INFORMATION;
```

Members

`EnlistmentId`

A GUID that KTM has assigned to the enlistment object.

`TransactionId`

A GUID that KTM has assigned to the transaction object that is associated with the enlistment object that the **EnlistmentId** member identifies.

`ResourceManagerId`

A GUID that KTM has assigned to the resource manager that created the enlistment.

Remarks

The **ENLISTMENT_BASIC INFORMATION** structure is used with the ZwQueryInformationEnlistment routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ENLISTMENT_INFORMATION_CLASS

ZwQueryInformationEnlistment

ENLISTMENT_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **ENLISTMENT_INFORMATION_CLASS** enumeration identifies the type of information that the ZwSetInformationEnlistment routine can set and that the ZwQueryInformationEnlistment routine can retrieve for an enlistment object.

Syntax

```
typedef enum _ENLISTMENT_INFORMATION_CLASS {
  EnlistmentBasicInformation,
  EnlistmentRecoveryInformation,
  EnlistmentCrmInformation
} ENLISTMENT_INFORMATION_CLASS;
```

Constants

EnlistmentBasicInformation	Information about an enlistment object is stored in an ENLISTMENT_BASIC_INFORMATION structure.
EnlistmentRecoveryInformation	A resource manager is setting or obtaining customized recovery information for an enlistment. To learn more about recovery information, see ZwSetInformationEnlistment.
EnlistmentCrmInformation	Information about an enlistment object is stored in an **ENLISTMENT_CRM_INFORMATION** structure.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ENLISTMENT_BASIC_INFORMATION

ZwQueryInformationEnlistment

ZwSetInformationEnlistment

EXT_DELETE_PARAMETERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **EXT_DELETE_PARAMETERS** structure contains an extended set of parameters for the ExDeleteTimer routine.

Syntax

```
typedef struct _EXT_DELETE_PARAMETERS {
  ULONG                Version;
  ULONG                Reserved;
  PEXT_DELETE_CALLBACK DeleteCallback;
  PVOID                DeleteContext;
} EXT_DELETE_PARAMETERS, *PEXT_DELETE_PARAMETERS;
```

Members

`Version`

The version number of this **EXT_DELETE_PARAMETERS** structure. The ExInitializeDeleteTimerParameters routine sets this member to the correct version number.

`Reserved`

Set to zero. The **ExInitializeDeleteTimerParameters** routine sets this member to zero.

`DeleteCallback`

A pointer to a driver-implemented ExTimerDeleteCallback callback routine. The operating system calls this routine when the timer is deleted. This parameter is optional and can be **NULL** if no timer-deletion callback routine is needed.

The **ExInitializeDeleteTimerParameters** routine sets this member to **NULL**. For more information, see Remarks.

`DeleteContext`

A context value for the timer-deletion callback routine. The operating system passes this value as a parameter to the timer-deletion callback routine, if one is specified. This parameter is typically a pointer to a caller-defined structure that contains context information used by the callback routine. This parameter is optional and can be set to **NULL** if no context information is needed.

The **ExInitializeDeleteTimerParameters** routine sets this member to **NULL**.

Remarks

The *Parameters* parameter of the ExDeleteTimer routine is a pointer to an **EXT_DELETE_PARAMETERS** structure. Before passing an **EXT_DELETE_PARAMETERS** structure to this routine, call the ExInitializeDeleteTimerParameters routine to initialize the structure.

Your driver can use an ExTimerDeleteCallback callback routine to free any storage or other system resources that the driver might have previously allocated to use with the timer object that is being deleted. For example, the driver's context value might point to a caller-allocated structure that needs to be freed if the timer object is to be

deleted. The operating system calls the *ExTimerDeleteCallback* routine only after the timer object has been disabled to prevent further timer operations and any pending timer operation on the timer object is canceled or completed.

Requirements

Minimum supported client	Supported starting with Windows 8.1.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ExDeleteTimer

ExInitializeDeleteTimerParameters

ExTimerDeleteCallback

EXT_SET_PARAMETERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **EXT_SET_PARAMETERS** structure contains an extended set of parameters for the ExSetTimer routine.

Syntax

```
typedef struct _EXT_SET_PARAMETERS_V0 {
  ULONG    Version;
  ULONG    Reserved;
  LONGLONG NoWakeTolerance;
} EXT_SET_PARAMETERS, *PEXT_SET_PARAMETERS;
```

Members

`Version`

The version number of this **EXT_SET_PARAMETERS** structure. The ExInitializeSetTimerParameters routine sets this member to the correct version number.

`Reserved`

Set to zero. The **ExInitializeSetTimerParameters** routine sets this member to zero.

`NoWakeTolerance`

The maximum time, in system time units (100-nanosecond intervals), that the timer can wait to wake the processor after the timer reaches its expiration time. Only after the processor wakes can the timer expire. If a timer is set to expire when the processor is in a low-power state, the timer will not wake the processor to expire until the expiration time plus the **NoWakeTolerance** delay is exceeded. As an option, a driver can set this member to EX_TIMER_UNLIMITED_TOLERANCE, which indicates that the timer never wakes the processor and, thus, cannot expire until the processor wakes for some other reason.

Do not set this member to a negative value (other than EX_TIMER_UNLIMITED_TOLERANCE). Otherwise, the routine bug checks.

Remarks

The *Parameters* parameter of the ExSetTimer routine is a pointer to an **EXT_SET_PARAMETERS** structure. Before passing an **EXT_SET_PARAMETERS** structure to this routine, call the ExInitializeSetTimerParameters routine to initialize the structure.

The **ExInitializeSetTimerParameters** routine sets the **NoWakeTolerance** member to zero, which means that the timer immediately wakes the processor if the timer reaches its expiration time when the processor is in a low-power state. To reduce power consumption, a driver can set this member to a nonzero value to extend the time that the processor spends in a low-power state.

Requirements

Minimum supported client	Supported starting with Windows 8.1.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ExInitializeSetTimerParameters

ExSetTimer

FILE_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_BASIC_INFORMATION** structure is used as an argument to routines that query or set file information.

Syntax

```
typedef struct _FILE_BASIC_INFORMATION {
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  ULONG         FileAttributes;
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

Members

`CreationTime`

Specifies the time that the file was created.

`LastAccessTime`

Specifies the time that the file was last accessed.

`LastWriteTime`

Specifies the time that the file was last written to.

`ChangeTime`

Specifies the last time the file was changed.

`FileAttributes`

Specifies one or more FILE_ATTRIBUTE_*XXX* flags. For descriptions of these flags, see the documentation for the **GetFileAttributes** function in the Microsoft Windows SDK.

Remarks

The FILE_ATTRIBUTE_NORMAL flag cannot be set or returned in combination with any other attributes. All other **FileAttributes** values override this attribute.

Time values **CreationTime**, **LastAccessTime**, **LastWriteTime**, and **ChangeTime** are expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

If you specify a value of zero for any of the *Xxx***Time** members of the **FILE_BASIC_INFORMATION** structure, the ZwSetInformationFile function keeps a file's current setting for that time.

The file system updates the values of the **LastAccessTime**, **LastWriteTime**, and **ChangeTime** members as appropriate after an I/O operation is performed on a file. However, a driver or application can request that the file system not update one or more of these members for I/O operations that are performed on the caller's file handle by setting the appropriate members to -1. The caller can set one, all, or any other combination of these three

members to -1. Only the members that are set to -1 will be unaffected by I/O operations on the file handle; the other members will be updated as appropriate.

To set the members of this structure, the caller must have FILE_WRITE_ATTRIBUTES access to the file.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KeQuerySystemTime

ZwCreateFile

ZwQueryInformationFile

ZwSetInformationFile

FILE_FS_DEVICE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_FS_DEVICE_INFORMATION** structure provides file system device information about the type of device object associated with a file object.

Syntax

```
typedef struct _FILE_FS_DEVICE_INFORMATION {
  DEVICE_TYPE DeviceType;
  ULONG       Characteristics;
} FILE_FS_DEVICE_INFORMATION, *PFILE_FS_DEVICE_INFORMATION;
```

Members

`DeviceType`

Set when a driver calls **IoCreateDevice** as appropriate for the type of underlying device. For more information, see Specifying Device Types.

`Characteristics`

The device characteristics. For a description of relevant values, see DEVICE_OBJECT.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DEVICE_OBJECT

FILE_FULL_EA_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_FULL_EA_INFORMATION** structure provides extended attribute (EA) information. This structure is used primarily by network drivers.

Syntax

```
typedef struct _FILE_FULL_EA_INFORMATION {
  ULONG  NextEntryOffset;
  UCHAR  Flags;
  UCHAR  EaNameLength;
  USHORT EaValueLength;
  CHAR   EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Members

`NextEntryOffset`

The offset of the next **FILE_FULL_EA_INFORMATION**-type entry. This member is zero if no other entries follow this one.

`Flags`

Can be zero or can be set with FILE_NEED_EA, indicating that the file to which the EA belongs cannot be interpreted without understanding the associated extended attributes.

`EaNameLength`

The length in bytes of the **EaName** array. This value does not include a null-terminator to **EaName**.

`EaValueLength`

The length in bytes of each EA value in the array.

`EaName`

An array of characters naming the EA for this entry.

Remarks

This structure is longword-aligned. If a set of **FILE_FULL_EA_INFORMATION** entries is buffered, **NextEntryOffset** value in each entry, except the last, falls on a longword boundary.

The value(s) associated with each entry follows the **EaName** array. That is, an EA's values are located at **EaName** + (**EaNameLength** + 1).

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwCreateFile

FILE_IO_PRIORITY_HINT_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_IO_PRIORITY_HINT_INFORMATION** structure is used by the ZwQueryInformationFile and ZwSetInformationFile routines to query and set the default IRP priority hint for requests on the specified file handle.

Syntax

```
typedef struct _FILE_IO_PRIORITY_HINT_INFORMATION {
  IO_PRIORITY_HINT PriorityHint;
} FILE_IO_PRIORITY_HINT_INFORMATION, *PFILE_IO_PRIORITY_HINT_INFORMATION;
```

Members

`PriorityHint`

Specifies the IO_PRIORITY_HINT value that indicates the priority hint for a file handle.

Remarks

For more information about priority hints, see Using IRP Priority Hints.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_PRIORITY_HINT

ZwQueryInformationFile

ZwSetInformationFile

FILE_IS_REMOTE_DEVICE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_IS_REMOTE_DEVICE_INFORMATION** structure is used as an argument to the ZwQueryInformationFile routine.

Syntax

```
typedef struct _FILE_IS_REMOTE_DEVICE_INFORMATION {
  BOOLEAN IsRemote;
} FILE_IS_REMOTE_DEVICE_INFORMATION, *PFILE_IS_REMOTE_DEVICE_INFORMATION;
```

Members

`IsRemote`

A value that indicates whether the file system that contains the file is a remote file system.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwQueryInformationFile

FILE_NETWORK_OPEN_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_NETWORK_OPEN_INFORMATION** structure is used as an argument to ZwQueryInformationFile.

Syntax

```
typedef struct _FILE_NETWORK_OPEN_INFORMATION {
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  LARGE_INTEGER AllocationSize;
  LARGE_INTEGER EndOfFile;
  ULONG         FileAttributes;
} FILE_NETWORK_OPEN_INFORMATION, *PFILE_NETWORK_OPEN_INFORMATION;
```

Members

`CreationTime`

Specifies the time that the file was created.

`LastAccessTime`

Specifies the time that the file was last accessed.

`LastWriteTime`

Specifies he time that the file was last written to.

`ChangeTime`

Specifies the time that the file was last changed.

`AllocationSize`

Specifies the file allocation size, in bytes. Usually, this value is a multiple of the sector or cluster size of the underlying physical device.

`EndOfFile`

Specifies the absolute end-of-file position as a byte offset from the start of the file. **EndOfFile** specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file. In other words, **EndOfFile** is the offset to the byte immediately following the last valid byte in the file.

`FileAttributes`

Specifies one or more FILE_ATTRIBUTE_*XXX* flags. For descriptions of these flags, see the documentation of the **GetFileAttributes** function in the Microsoft Windows SDK.

Remarks

FILE_READ_ATTRIBUTES access to the file is required to query this information.

Time values **CreationTime**, **LastAccessTime**, **LastWriteTime**, and **ChangeTime** are expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

This structure must be aligned on a LONGLONG (8-byte) boundary.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwQueryInformationFile

FILE_OBJECT structure

1/30/2019 • 10 minutes to read • Edit Online

The **FILE_OBJECT** structure is used by the system to represent a file object. To user-mode protected subsystems, a file object represents an open instance of a file, device, directory, or volume. To device and intermediate drivers, a file object usually represents a device object. To drivers in the file system stack, a file object usually represents a directory or file.

A file object is partially opaque. Certain types of drivers, such as file system drivers and network transport drivers, use some of the fields of file objects.

Syntax

```
typedef struct _FILE_OBJECT {
  CSHORT                          Type;
  CSHORT                          Size;
  PDEVICE_OBJECT                  DeviceObject;
  PVPB                            Vpb;
  PVOID                           FsContext;
  PVOID                           FsContext2;
  PSECTION_OBJECT_POINTERS        SectionObjectPointer;
  PVOID                           PrivateCacheMap;
  NTSTATUS                        FinalStatus;
  struct _FILE_OBJECT             *RelatedFileObject;
  BOOLEAN                         LockOperation;
  BOOLEAN                         DeletePending;
  BOOLEAN                         ReadAccess;
  BOOLEAN                         WriteAccess;
  BOOLEAN                         DeleteAccess;
  BOOLEAN                         SharedRead;
  BOOLEAN                         SharedWrite;
  BOOLEAN                         SharedDelete;
  ULONG                           Flags;
  UNICODE_STRING                  FileName;
  LARGE_INTEGER                   CurrentByteOffset;
  __volatile ULONG                Waiters;
  __volatile ULONG                Busy;
  PVOID                           LastLock;
  KEVENT                          Lock;
  KEVENT                          Event;
  __volatile PIO_COMPLETION_CONTEXT CompletionContext;
  KSPIN_LOCK                      IrpListLock;
  LIST_ENTRY                      IrpList;
  __volatile PVOID                FileObjectExtension;
} FILE_OBJECT, *PFILE_OBJECT;
```

Members

`Type`

A read-only member used by the system to indicate that the object is a file object. If the object is a file object, the value of this member is 5.

`Size`

A read-only member that specifies the size, in bytes, of the file object. This size does not include the file object extension, if one is present.

`DeviceObject`

A pointer to the device object on which the file is opened.

`Vpb`

A pointer to the volume parameter block associated with the file object.

Note that if the **Vpb** member is non-**NULL**, the file resides on a mounted volume.

`FsContext`

A pointer to whatever optional state a driver maintains about the file object; otherwise, **NULL**. For file system drivers, this member must point to a FSRTL_ADVANCED_FCB_HEADER header structure that is contained within a file-system-specific structure; otherwise system instability can result. Usually, this header structure is embedded in a file control block (FCB). However, on some file systems that support multiple data streams, such as NTFS, this header structure is a stream control block (SCB).

> **Note** In a WDM device stack, only the functional device object (FDO) can use the two context pointers. File system drivers share this member across multiple opens to the same data stream.

`FsContext2`

A pointer to whatever additional state a driver maintains about the file object; otherwise, **NULL**.

> **Note** This member is opaque for drivers in the file system stack because the underlying file system utilizes this member.

`SectionObjectPointer`

A pointer to the file object's read-only section object. This member is set only by file systems and used for Cache Manager interaction.

`PrivateCacheMap`

An opaque member, set only by file systems, that points to handle-specific information and that is used for Cache Manager interaction.

`FinalStatus`

A read-only member that is used, in certain synchronous cases, to indicate the final status of the file object's I/O request.

`RelatedFileObject`

A pointer to a **FILE_OBJECT** structure used to indicate that the current file object has been opened relative to an already open file object. The file object pointed to by this member is usually a directory (meaning the current file has been opened relative to this directory). However, a file can be reopened relative to itself, and alternate data streams for a file can be opened relative to an already open primary data stream for that same file. The **RelatedFileObject** member is only valid during the processing of the IRP_MJ_CREATE requests.

`LockOperation`

A read-only member. If **FALSE**, a lock operation (**NtLockFile**) has never been performed on the file object. If **TRUE**, at least one lock operation has been performed on the file object. Once set to **TRUE**, this member always remains **TRUE** (for example, releasing file locks on the file object does not reset this member to **FALSE**).

`DeletePending`

A read-only member. If **TRUE**, a delete operation for the file associated with the file object exists. If **FALSE**, there currently is no pending delete operation for the file object.

`ReadAccess`

A read-only member. If **TRUE**, the file associated with the file object has been opened for read access. If **FALSE**, the file has been opened without read access. This information is used when checking and/or setting the share access of the file.

`WriteAccess`

A read-only member. If **TRUE**, the file associated with the file object has been opened for write access. If **FALSE**, the file has been opened without write access. This information is used when checking and/or setting the share access of the file.

`DeleteAccess`

A read-only member. If **TRUE**, the file associated with the file object has been opened for delete access. If **FALSE**, the file has been opened without delete access. This information is used when checking and/or setting the share access of the file.

`SharedRead`

A read-only member. If **TRUE**, the file associated with the file object has been opened for read sharing access. If **FALSE**, the file has been opened without read sharing access. This information is used when checking and/or setting the share access of the file.

`SharedWrite`

A read-only member. If **TRUE**, the file associated with the file object has been opened for write sharing access. If **FALSE**, the file has been opened without write sharing access. This information is used when checking and/or setting the share access of the file.

`SharedDelete`

A read-only member. If **TRUE**, the file associated with the file object has been opened for delete sharing access. If **FALSE**, the file has been opened without delete sharing access. This information is used when checking and/or setting the share access of the file.

`Flags`

A read-only member used by the system to hold one or more (a bitwise inclusive OR combination) of the following private flag values.

FLAG	MEANING
FO_FILE_OPEN	Deprecated.
FO_SYNCHRONOUS_IO	The file object is opened for synchronous I/O.
FO_ALERTABLE_IO	Any wait in the I/O manager, as a result of a request made to this file object, is alertable.
FO_NO_INTERMEDIATE_BUFFERING	The file associated with the file object cannot be cached or buffered in a driver's internal buffers.
FO_WRITE_THROUGH	System services, file system drivers, and drivers that write data to the file must transfer the data into the file before any requested write operation is considered complete.

FO_SEQUENTIAL_ONLY	The file associated with the file object was opened for sequential I/O operations only.
FO_CACHE_SUPPORTED	The file associated with the file object is cacheable. This flag should be set only by a file system driver, and only if the **FsContext** member points to a valid FSRTL_ADVANCED_FCB_HEADER structure.
FO_NAMED_PIPE	The file object represents a named pipe.
FO_STREAM_FILE	The file object represents a file stream.
FO_MAILSLOT	The file object represents a mailslot.
FO_GENERATE_AUDIT_ON_CLOSE	Deprecated.
FO_QUEUE_IRP_TO_THREAD	IRPs will not be queued to this file object.
FO_DIRECT_DEVICE_OPEN	The device targeted by this file object was opened directly.
FO_FILE_MODIFIED	The file associated with the file object has been modified.
FO_FILE_SIZE_CHANGED	The file associated with the file object has changed in size.
FO_CLEANUP_COMPLETE	The file system has completed its cleanup for this file object.
FO_TEMPORARY_FILE	The file associated with the file object is a temporary file.
FO_DELETE_ON_CLOSE	The file associated with the file object will be deleted by the file system upon close.
FO_OPENED_CASE_SENSITIVE	The file name case of the file associated with the file object is respected.
FO_HANDLE_CREATED	A file handle was created for file object.
FO_FILE_FAST_IO_READ	A fast I/O read was performed on this file object.
FO_RANDOM_ACCESS	The file associated with the file object was opened for random access.
FO_FILE_OPEN_CANCELLED	The create request for this file object was canceled before completing.
FO_VOLUME_OPEN	The file object represents a volume open request.
FO_REMOTE_ORIGIN	The create request for the file associated with the file object originated on a remote machine.
FO_SKIP_COMPLETION_PORT	For a file object associated with a port, determines if the system should skip queuing to the completion port when the IRP is completed synchronously with a non-error status return value.

FO_SKIP_SET_EVENT	Skip setting the event for the file object upon IRP completion.
FO_SKIP_SET_FAST_IO	Skip setting an event supplied to a system service when the fast I/O path is successful.

`FileName`

A UNICODE_STRING structure whose **Buffer** member points to a read-only Unicode string that holds the name of the file opened on the volume. If the volume is being opened, the **Length** member of the **UNICODE_STRING** structure will be zero. Note that the file name in this string is valid only during the initial processing of an IRP_MJ_CREATE request. This file name should not be considered valid after the file system starts to process the **IRP_MJ_CREATE** request. The storage for the string pointed to by the **Buffer** member of the **UNICODE_STRING** structure is allocated in paged system memory. For more information about obtaining a file name, see FltGetFileNameInformation.

`CurrentByteOffset`

A read-only member that specifies the file offset, in bytes, associated with the file object.

`Waiters`

A read-only member used by the system to count the number of outstanding waiters on a file object opened for synchronous access.

`Busy`

A read-only member used by the system to indicate whether a file object opened for synchronous access is currently busy.

`LastLock`

An opaque pointer to the last lock applied to the file object.

`Lock`

An opaque member used by the system to hold a file object event lock. The event lock is used to control synchronous access to the file object. Applicable only to file objects that are opened for synchronous access.

`Event`

An opaque member used by the system to hold an event object for the file object. The event object is used to signal the completion of an I/O request on the file object if no user event was supplied or a synchronous API was called.

`CompletionContext`

An opaque pointer to completion port information (port pointer and key) associated with the file object, if any.

`IrpListLock`

An opaque pointer to a KSPIN_LOCK structure that serves as the spin lock used to synchronize access to the file object's IRP list.

`IrpList`

An opaque pointer to the head of the IRP list associated with the file object.

`FileObjectExtension`

An opaque pointer to the file object's file object extension (FOBX) structure. The **FOBX** structure contains various opaque contexts used internally as well as the per-file object contexts available through **FsRtl*Xxx*** routines.

Remarks

Drivers can use the **FsContext** and **FsContext2** members to maintain driver-determined state about an open file object. A driver cannot use these members unless the file object is accessible in the driver's I/O stack location of an IRP.

All remaining members in a file object are either opaque or read-only:

- Opaque members within a file object should be considered inaccessible. Drivers with dependencies on object field locations or access to opaque members might not remain portable and interoperable with other drivers over time.
- Drivers can use read-only members to acquire relevant information but must not modify read-only members and, if a pointer, the object that the member points to.

During the processing of an IRP_MJ_CREATE request, a file system driver calls the IoSetShareAccess routine (if the client is the first to open the file) or the IoCheckShareAccess routine (for subsequent clients that want to share the file). **IoSetShareAccess** and **IoCheckShareAccess** update the **ReadAccess**, **WriteAccess**, and **DeleteAccess** members to indicate the access rights that are granted to the client if the client has exclusive access to the file. Additionally, **IoCheckShareAccess** updates the **SharedRead**, **SharedWrite**, and **SharedDelete** members to indicate the access rights that are simultaneously granted to two or more clients that share the file. If the driver for a device other than a file system has to monitor the access rights of clients, this driver typically stores access rights information in context buffers that are pointed to by the **FsContext** and **FsContext2** members.

> **Note** The type of object (for example, a file, directory, or volume) that a given file object represents cannot be determined by only examining the contents of the file object structure. For information about how to determine the type of object that a file object represents, see ZwQueryInformationFile.

The Common Log File System (CLFS) uses the **LOG_FILE_OBJECT** structure to represent logs. The ClfsCreateLogFile function returns a pointer to a **LOG_FILE_OBJECT** structure, which clients then pass to other CLFS functions. CLFS clients do not directly access the members of a **LOG_FILE_OBJECT** structure.

C++

```
typedef FILE_OBJECT LOG_FILE_OBJECT, *PLOG_FILE_OBJECT, **PPLOG_FILE_OBJECT;
```

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

DEVICE_OBJECT

FOBX

FSRTL_ADVANCED_FCB_HEADER

IRP_MJ_CREATE

IoCheckShareAccess

IoGetDeviceObjectPointer

IoSetShareAccess

ObDereferenceObject

ZwQueryInformationFile

FILE_POSITION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_POSITION_INFORMATION** structure is used as an argument to routines that query or set file information.

Syntax

```
typedef struct _FILE_POSITION_INFORMATION {
  LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION, *PFILE_POSITION_INFORMATION;
```

Members

`CurrentByteOffset`

The byte offset of the current file pointer.

Remarks

FILE_READ_DATA or FILE_WRITE_DATA access to the file is required to change this information about the file, and the file must be opened for synchronous I/O.

If the file was opened or created with the FILE_NO_INTERMEDIATE_BUFFERING option, the value of **CurrentByteOffset** must be an integral multiple of the sector size of the underlying device.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwCreateFile

ZwSetInformationFile

FILE_STANDARD_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_STANDARD_INFORMATION** structure is used as an argument to routines that query or set file information.

Syntax

```
typedef struct _FILE_STANDARD_INFORMATION {
  LARGE_INTEGER AllocationSize;
  LARGE_INTEGER EndOfFile;
  ULONG        NumberOfLinks;
  BOOLEAN      DeletePending;
  BOOLEAN      Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

Members

`AllocationSize`

The file allocation size in bytes. Usually, this value is a multiple of the sector or cluster size of the underlying physical device.

`EndOfFile`

The end of file location as a byte offset.

`NumberOfLinks`

The number of hard links to the file.

`DeletePending`

The delete pending status. **TRUE** indicates that a file deletion has been requested.

`Directory`

The file directory status. **TRUE** indicates the file object represents a directory.

Remarks

EndOfFile specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file; that is, it is the offset to the byte immediately following the last valid byte in the file.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwCreateFile

ZwQueryInformationFile

ZwSetInformationFile

FILE_STANDARD_INFORMATION_EX structure

1/30/2019 • 2 minutes to read • Edit Online

The **FILE_STANDARD_INFORMATION_EX** structure is used as an argument to routines that query or set file information.

Syntax

```
typedef struct _FILE_STANDARD_INFORMATION_EX {
  LARGE_INTEGER AllocationSize;
  LARGE_INTEGER EndOfFile;
  ULONG         NumberOfLinks;
  BOOLEAN       DeletePending;
  BOOLEAN       Directory;
  BOOLEAN       AlternateStream;
  BOOLEAN       MetadataAttribute;
} FILE_STANDARD_INFORMATION_EX, *PFILE_STANDARD_INFORMATION_EX;
```

Members

`AllocationSize`

The file allocation size in bytes. Usually, this value is a multiple of the sector or cluster size of the underlying physical device.

`EndOfFile`

The end of file location as a byte offset.

`NumberOfLinks`

The number of hard links to the file.

`DeletePending`

The delete pending status. **TRUE** indicates that a file deletion has been requested.

`Directory`

The file directory status. **TRUE** indicates the file object represents a directory.

`AlternateStream`

The alternate data stream status. **TRUE** indicates the file object represents an alternate data stream.

`MetadataAttribute`

The metadata attribute status. **TRUE** indicates the file object represents a metadata attribute.

Remarks

EndOfFile specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file; that is, it is the offset to the byte immediately following the last valid byte in the file.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwCreateFile

ZwQueryInformationFile

ZwSetInformationFile

FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **FUNCTION_LEVEL_DEVICE_RESET_PARAMETER** structure is used as an argument to the DeviceReset routine of the GUID_DEVICE_RESET_INTERFACE_STANDARD interface. This structure specifies a callback routine that is called when a function-level device reset is completed, and a context structure that is passed to the callback routine. For more information, see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD

Syntax

```
typedef struct _FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS {
  ULONG                    Size;
  PDEVICE_RESET_COMPLETION DeviceResetCompletion;
  PVOID                    CompletionContext;
} FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS, *PFUNCTION_LEVEL_DEVICE_RESET_PARAMETERS;
```

Members

`Size`

The size, in bytes, of this structure.

`DeviceResetCompletion`

Pointer to a completion callback routine to be called when a function-level device reset is completed. The function prototype for this callback routine is defined as follows:

```
typedef
VOID
(*PDEVICE_RESET_COMPLETION)(
    _In_ NTSTATUS Status,
    _Inout_opt_ PVOID Context
    );
```

`CompletionContext`

Points to a caller-supplied context structure to be passed to the *DeviceResetCompletion* callback.

Requirements

Header	wdm.h

See Also

DEVICE_RESET_INTERFACE_STANDARD

DeviceReset

Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD

GENERIC_MAPPING structure

1/30/2019 • 2 minutes to read • Edit Online

The **GENERIC_MAPPING** structure describes the ACCESS_MASK value of specific access rights associated with each type of generic access right.

Syntax

```
typedef struct _GENERIC_MAPPING {
  ACCESS_MASK GenericRead;
  ACCESS_MASK GenericWrite;
  ACCESS_MASK GenericExecute;
  ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

Members

`GenericRead`

Describes the specific access rights corresponding to the GENERIC_READ access right.

`GenericWrite`

Describes the specific access rights corresponding to the GENERIC_WRITE access right.

`GenericExecute`

Describes the specific access rights corresponding to the GENERIC_EXECUTE access right.

`GenericAll`

Describes the specific access rights corresponding to the GENERIC_ALL access right.

Remarks

Use the IoGetFileObjectGenericMapping routine to get the specific access rights corresponding to generic access rights for file objects. Use RtlMapGenericMask to subtract off the access rights within an ACCESS_MASK that do not derive from generic access rights.

For more information about generic access rights, see the reference page for **GENERIC_MAPPING** in the Microsoft Windows SDK documentation.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

IoGetFileObjectGenericMapping

RtlMapGenericMask

HWPROFILE_CHANGE_NOTIFICATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **HWPROFILE_CHANGE_NOTIFICATION** structure describes an event related to a hardware profile configuration change. The PnP manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryHardwareProfileChange** events.

Syntax

```
typedef struct _HWPROFILE_CHANGE_NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID   Event;
} HWPROFILE_CHANGE_NOTIFICATION, *PHWPROFILE_CHANGE_NOTIFICATION;
```

Members

`Version`

Specifies the version of the data structure, currently 1.

`Size`

Specifies the size of the structure, in bytes including the size of the standard first three members plus the event-specific data.

`Event`

Specifies a GUID identifying the event: GUID_HWPROFILE_QUERY_CHANGE, GUID_HWPROFILE_CHANGE_COMPLETE, or GUID_HWPROFILE_CHANGE_CANCELLED. The GUIDs are defined in Wdmguid.h.

Remarks

There is no event-specific data for a hardware-profile-change event.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION

IoRegisterPlugPlayNotification

PLUGPLAY_NOTIFICATION_HEADER

TARGET_DEVICE_REMOVAL_NOTIFICATION

IMAGE_POLICY_ENTRY structure

1/30/2019 • 2 minutes to read • Edit Online

This structure is not supported.

Syntax

```
typedef struct _IMAGE_POLICY_ENTRY {
  IMAGE_POLICY_ENTRY_TYPE Type;
  IMAGE_POLICY_ID         PolicyId;
  union {
    const VOID *None;
    BOOLEAN    BoolValue;
    INT8       Int8Value;
    UINT8      UInt8Value;
    INT16      Int16Value;
    UINT16     UInt16Value;
    INT32      Int32Value;
    UINT32     UInt32Value;
    INT64      Int64Value;
    UINT64     UInt64Value;
    PCSTR      AnsiStringValue;
    PCWSTR     UnicodeStringValue;
  } u;
} IMAGE_POLICY_ENTRY;
```

Members

`Type`

`PolicyId`

`u`

`u.None`

`u.BoolValue`

`u.Int8Value`

`u.UInt8Value`

`u.Int16Value`

`u.UInt16Value`

`u.Int32Value`

`u.UInt32Value`

`u.Int64Value`

`u.UInt64Value`

`u.AnsiStringValue`

`u.UnicodeStringValue`

Requirements

Header	wdm.h

IMAGE_POLICY_ENTRY_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

This enumeration is not supported.

Syntax

```
typedef enum _IMAGE_POLICY_ENTRY_TYPE {
  ImagePolicyEntryTypeNone,
  ImagePolicyEntryTypeBool,
  ImagePolicyEntryTypeInt8,
  ImagePolicyEntryTypeUInt8,
  ImagePolicyEntryTypeInt16,
  ImagePolicyEntryTypeUInt16,
  ImagePolicyEntryTypeInt32,
  ImagePolicyEntryTypeUInt32,
  ImagePolicyEntryTypeInt64,
  ImagePolicyEntryTypeUInt64,
  ImagePolicyEntryTypeAnsiString,
  ImagePolicyEntryTypeUnicodeString,
  ImagePolicyEntryTypeOverride,
  ImagePolicyEntryTypeMaximum
} IMAGE_POLICY_ENTRY_TYPE;
```

Constants

ImagePolicyEntryTypeNone	
ImagePolicyEntryTypeBool	Reserved.
ImagePolicyEntryTypeInt8	Reserved.
ImagePolicyEntryTypeUInt8	Reserved.
ImagePolicyEntryTypeInt16	Reserved.
ImagePolicyEntryTypeUInt16	Reserved.
ImagePolicyEntryTypeInt32	Reserved.
ImagePolicyEntryTypeUInt32	Reserved.
ImagePolicyEntryTypeInt64	Reserved.
ImagePolicyEntryTypeUInt64	Reserved.
ImagePolicyEntryTypeAnsiString	Reserved.
ImagePolicyEntryTypeUnicodeString	Reserved.

ImagePolicyEntryTypeOverride	Reserved.
ImagePolicyEntryTypeMaximum	Reserved.

Requirements

Header	wdm.h (include Wdm.h)

IMAGE_POLICY_ID Enumeration

1/30/2019 • 2 minutes to read • Edit Online

This enumeration is not supported.

Syntax

```
typedef enum _IMAGE_POLICY_ID {
  ImagePolicyIdNone,
  ImagePolicyIdEtw,
  ImagePolicyIdDebug,
  ImagePolicyIdCrashDump,
  ImagePolicyIdCrashDumpKey,
  ImagePolicyIdCrashDumpKeyGuid,
  ImagePolicyIdParentSd,
  ImagePolicyIdParentSdRev,
  ImagePolicyIdSvn,
  ImagePolicyIdDeviceId,
  ImagePolicyIdCapability,
  ImagePolicyIdScenarioId,
  ImagePolicyIdMaximum
} IMAGE_POLICY_ID;
```

Constants

ImagePolicyIdNone	
ImagePolicyIdEtw	Reserved.
ImagePolicyIdDebug	Reserved.
ImagePolicyIdCrashDump	Reserved.
ImagePolicyIdCrashDumpKey	Reserved.
ImagePolicyIdCrashDumpKeyGuid	Reserved.
ImagePolicyIdParentSd	Reserved.
ImagePolicyIdParentSdRev	Reserved.
ImagePolicyIdSvn	Reserved.
ImagePolicyIdDeviceId	Reserved.
ImagePolicyIdCapability	Reserved.
ImagePolicyIdScenarioId	Reserved.
ImagePolicyIdMaximum	Reserved.

Requirements

Header	wdm.h (include Wdm.h)

IMAGE_POLICY_METADATA structure

1/30/2019 • 2 minutes to read • Edit Online

This structure is not supported.

Syntax

```
typedef struct _IMAGE_POLICY_METADATA {
  UCHAR              Version;
  UCHAR              Reserved0[7];
  ULONGLONG          ApplicationId;
  IMAGE_POLICY_ENTRY Policies[];
} IMAGE_POLICY_METADATA;
```

Members

`Version`

`Reserved0`

`ApplicationId`

`Policies`

Requirements

Header	wdm.h (include Wdm.h, Wdm.h)

INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **INTERFACE** structure describes an interface that is exported by a driver for use by other drivers.

Syntax

```
typedef struct _INTERFACE {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
} INTERFACE, *PINTERFACE;
```

Members

`Size`

Size, in bytes, of a structure defining a driver interface, including this structure and interface-specific members.

`Version`

Driver-defined interface version.

`Context`

Pointer to interface-specific context information.

`InterfaceReference`

Pointer to a driver-supplied InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

Pointer to a driver-supplied InterfaceDereference routine that decrements the interface's reference count.

Remarks

The **INTERFACE** structure must be included as the first member of all structures that describe interfaces returned by drivers in response to an IRP_MN_QUERY_INTERFACE request.

The *InterfaceReference* routine must be called by the driver that exports the interface, each time the driver supplies that interface in response to **IRP_MN_QUERY_INTERFACE**. Likewise, if the driver that requests the interface subsequently passes it to another driver, the driver that passes the interface must call *InterfaceReference* on behalf of the driver that receives it.

Each driver that imports the interface (whether by sending **IRP_MN_QUERY_INTERFACE** or by receiving the interface from another driver) must call the *InterfaceDereference* routine after it has finished using the interface. After calling the *InterfaceDereference* routine, a driver cannot use the interface again without first reobtaining it.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

INTERFACE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **INTERFACE_TYPE** enumeration indicates the bus type.

Syntax

```
typedef enum _INTERFACE_TYPE {
  InterfaceTypeUndefined,
  Internal,
  Isa,
  Eisa,
  MicroChannel,
  TurboChannel,
  PCIBus,
  VMEBus,
  NuBus,
  PCMCIABus,
  CBus,
  MPIBus,
  MPSABus,
  ProcessorInternal,
  InternalPowerBus,
  PNPISABus,
  PNPBus,
  Vmcs,
  ACPIBus,
  MaximumInterfaceType
} INTERFACE_TYPE, *PINTERFACE_TYPE;
```

Constants

InterfaceTypeUndefined	Indicates that the interface type is undefined.
Internal	For internal use only.
Isa	Indicates that the interface is published by the ISA bus driver.
Eisa	Indicates that the interface is published by the EISA bus driver.
MicroChannel	Indicates that the interface is published by the MicroChannel bus driver.
TurboChannel	Indicates that the interface is published by the TurboChannel bus driver.
PCIBus	Indicates that the interface is published by the PCI bus driver.
VMEBus	Indicates that the interface is published by the VME bus driver.
NuBus	Indicates that the interface is published by the NuBus driver.

PCMCIABus	Indicates that the interface is published by the PCMCIA bus driver.
CBus	Indicates that the interface is published by the Cbus driver.
MPIBus	Indicates that the interface is published by the MPI bus driver.
MPSABus	Indicates that the interface is published by the MPSA bus driver.
ProcessorInternal	Indicates that the interface is published by the ISA bus driver.
InternalPowerBus	Indicates that the interface is published for an internal power bus. Some devices have power control ports that allow them to share power control with other devices. The Windows architecture represents these devices as slots on a virtual bus called an "internal power bus."
PNPISABus	Indicates that the interface is published by the PNPISA bus driver.
PNPBus	Indicates that the interface is published by the PNP bus driver.
Vmcs	Reserved for use by the operating system.
ACPIBus	Indicates that the interface is published by the ACPI bus driver. The ACPI bus driver enumerates devices that are described in the ACPI firmware of the hardware platform. These devices might physically reside on buses that are controlled by other bus drivers, but the ACPI bus driver must enumerate these devices because the other bus drivers cannot detect them. This interface type is defined starting with Windows 8.
MaximumInterfaceType	Marks the upper limit of the possible bus types.

Requirements

Header	wdm.h (include Wdm.h, Miniport.h, Wudfwdm.h)

See Also

HW_INITIALIZATION_DATA

PORT_CONFIGURATION_INFORMATION

ScsiPortGetDeviceBase

ScsiPortValidateRange

StorPortGetDeviceBase

StorPortValidateRange

IO_ACCESS_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Defines the types of access mode for Scheduled File I/O (SFIO).

Syntax

```
typedef enum _IO_ACCESS_MODE {
  SequentialAccess,
  RandomAccess
} IO_ACCESS_MODE;
```

Constants

SequentialAccess	Indicates that the input/output will be sent down in a sequential order.
RandomAccess	Indicates that the input/output might not be in a predictable order.

Requirements

Header	wdm.h (include Wdm.h, Ntddsfio.h)

IO_ACCESS_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Defines the access rights for Scheduled File I/O (SFIO).

Syntax

```
typedef enum _IO_ACCESS_TYPE {
  ReadAccess,
  WriteAccess,
  ModifyAccess
} IO_ACCESS_TYPE;
```

Constants

ReadAccess	Indicates that the input/output will be comprised solely of reads.
WriteAccess	Indicates that the input/output will be comprised solely of writes.
ModifyAccess	Indicates that the input/output will be comprised of reads and writes.

Requirements

Header	wdm.h (include Ntddsfio.h)

IO_ALLOCATION_ACTION Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_ALLOCATION_ACTION** enumerated type is used to specify return values for AdapterControl and ControllerControl routines.

Syntax

```
typedef enum _IO_ALLOCATION_ACTION {
  KeepObject,
  DeallocateObject,
  DeallocateObjectKeepRegisters
} IO_ALLOCATION_ACTION, *PIO_ALLOCATION_ACTION;
```

Constants

KeepObject	Indicates that you want the driver to retain ownership of the adapter or controller object.
DeallocateObject	Indicates that you do not want the driver to retain ownership of the adapter or controller object.
DeallocateObjectKeepRegisters	For adapter objects only. Indicates that you do not want the driver to retain ownership of the adapter object, but you do want the driver to retain ownership of the allocated map registers.

Remarks

If an *AdapterControl* or *ControllerControl* routine completes an IRP, or if it can set up an operation (such as a disk seek) for a target device object that could be overlapped with an operation for another device object, it should return **DeallocateObject**.

If a driver uses packet-based bus-master DMA, its *AdapterControl* routine should return **DeallocateObjectKeepRegisters**.

Otherwise, the driver should return **KeepObject**.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h)

IO_CONNECT_INTERRUPT_PARAMETERS structure

2/5/2019 • 7 minutes to read • Edit Online

The **IO_CONNECT_INTERRUPT_PARAMETERS** structure contains the parameters that a driver supplies to the IoConnectInterruptEx routine to register an interrupt service routine (ISR).

Syntax

```
typedef struct _IO_CONNECT_INTERRUPT_PARAMETERS {
  ULONG Version;
  union {
    IO_CONNECT_INTERRUPT_FULLY_SPECIFIED_PARAMETERS FullySpecified;
    IO_CONNECT_INTERRUPT_LINE_BASED_PARAMETERS      LineBased;
    IO_CONNECT_INTERRUPT_MESSAGE_BASED_PARAMETERS   MessageBased;
  };
} IO_CONNECT_INTERRUPT_PARAMETERS, *PIO_CONNECT_INTERRUPT_PARAMETERS;
```

Members

`Version`

On input, specifies the particular operation to be performed by **IoConnectInterruptEx**, as follows.

VERSION VALUE	IOCONNECTINTERRUPTEX OPERATION
CONNECT_FULLY_SPECIFIED	Connects to a specific interrupt using information provided by the Plug and Play (PnP) manager. Use the **FullySpecified** member to provide the additional parameters of the operation.
CONNECT_LINE_BASED	Registers an InterruptService routine for the device's line-based interrupts. Use the **LineBased** member to provide the additional parameters of the operation.
CONNECT_MESSAGE_BASED	Registers an InterruptMessageService routine for the device's message-signaled interrupts. The caller can also specify a fallback *InterruptService* routine if the device only has line-based interrupts. Use the **MessageBased** member to provide the additional parameters of the operation.

On return, the routine provides information about the operation, as follows.

VERSION VALUE	DESCRIPTION
CONNECT_FULLY_SPECIFIED	The caller specified CONNECT_LINE_BASED or CONNECT_MESSAGE_BASED for **Version** on a platform that does not support it. Retry the operation using CONNECT_FULLY_SPECIFIED.
CONNECT_LINE_BASED	The caller specified CONNECT_MESSAGE_BASED and the caller's fallback *InterruptService* routine was registered.

CONNECT_MESSAGE_BASED	The caller specified CONNECT_MESSAGE_BASED and the caller's *InterruptMessageService* routine was registered.

`FullySpecified`

Specifies the additional parameters of the operation to be performed by IoConnectInterruptEx when **Version** has a value of CONNECT_FULLY_SPECIFIED (or CONNECT_FULLY_SPECIFIED_GROUP if the **Group** member is used). For more information, see Using the CONNECT_FULLY_SPECIFIED Version of IoConnectInterruptEx.

PhysicalDeviceObject

A pointer to the PDO for the device.

InterruptObject

A pointer to a location that receives a pointer to the set of interrupt objects for the device.

ServiceRoutine

A pointer to the InterruptService routine to register as the ISR for the device's interrupts.

ServiceContext

Specifies the value to be passed as the *ServiceContext* parameter of the InterruptService routine.

SpinLock

Either a pointer to a spin lock to serve as the interrupt spin lock for the set of interrupts, or **NULL**. If **NULL**, the system allocates a spin lock to serve as the interrupt spin lock. If non-**NULL**, you should have initialized the spin lock with KeInitializeSpinLock.

SynchronizeIrql

Specifies the DIRQL at which the ISR will run. If the ISR handles more than one interrupt vector or the driver has more than one ISR, this value must be the maximum IRQL of the set of interrupts. The IRQL for an interrupt is passed in the **CmResourceTypeInterrupt** resource at the **u.Interrupt.Level** member of CM_PARTIAL_RESOURCE_DESCRIPTOR. Otherwise, the **Irql** and **SynchronizeIrql** values are identical.

FloatingSave

Specifies if the system saves the processor's floating-point state when the interrupt occurs. If **TRUE**, the system saves the floating-point state.

ShareVector

Specifies whether the interrupt vector is sharable. Line-based PCI interrupts must be sharable. For message-signaled PCI interrupts, driver writers can choose whether their interrupts are sharable, but should choose to make them sharable by default.

Vector

Specifies the interrupt vector passed in the **CmResourceTypeInterrupt** resource at the **u.Interrupt.Vector** member of **CM_PARTIAL_RESOURCE_DESCRIPTOR**.

Irql

Specifies the DIRQL passed in the **CmResourceTypeInterrupt** resource at the **u.Interrupt.Level** member of CM_PARTIAL_RESOURCE_DESCRIPTOR. Starting with Windows 8, a driver can register an ISR that runs at passive level by setting **Irql** and **SynchronizeIrql** to PASSIVE_LEVEL, and setting **SpinLock** to **NULL**. For more information, see Using Passive-Level Interrupt Service Routines.

InterruptMode

Specifies a KINTERRUPT_MODE that determines whether the interrupt is level-triggered (**InterruptMode** = **LevelSensitive**) or edge-triggered (**InterruptMode** = **Latched**). For shared interrupt lines from a PCI bus, specify **LevelSensitive**. For PCI message-signaled interrupts, specify **Latched**.

ProcessorEnableMask

Specifies the **KAFFINITY** mask that represents the set of processors on which the device's interrupts can occur. This value is passed in the **CmResourceTypeInterrupt** resource at the **u.Interrupt.Affinity** member of

CM_PARTIAL_RESOURCE_DESCRIPTOR.

Group

Specifies a group number that identifies the processor group to which the interrupt is to be delivered. Typically, a driver receives its group number as part of the translated resources that are included in an IRP_MN_START_DEVICE request. Starting with Windows 7, the **Group** member is used if the **Version** member of the **IO_CONNECT_INTERRUPT_PARAMETERS** structure is set to CONNECT_FULLY_SPECIFIED_GROUP. The **Group** member is ignored if **Version** is set to CONNECT_FULLY_SPECIFIED, in which case the group number for delivery of the interrupt is always 0.

`LineBased`

Specifies the additional parameters of the operation to be performed by IoConnectInterruptEx when **Version** has a value of CONNECT_LINE_BASED. For more information, see Using the CONNECT_LINE_BASED Version of IoConnectInterruptEx.

PhysicalDeviceObject

A pointer to the physical device object (PDO) of the device.

InterruptObject

A pointer to a location that receives a pointer to the set of interrupt objects for the device.

ServiceRoutine

A pointer to the InterruptService routine to register as the ISR for the device's interrupts.

ServiceContext

Specifies the value to be passed as the *ServiceContext* parameter of the *InterruptService* routine.

SpinLock

Either a pointer to a spin lock to serve as the interrupt spin lock for the set of interrupts, or **NULL**. If **NULL**, the system allocates a spin lock to serve as the interrupt spin lock. If non-**NULL**, you should have initialized the spin lock with KeInitializeSpinLock.

SynchronizeIrql

Specifies the minimum device IRQL (DIRQL) at which the ISR runs. The system uses this value only if it is greater than the maximum IRQL of the set of interrupts; otherwise, the system uses the maximum IRQL. Drivers almost always specify PASSIVE_LEVEL for **SynchronizeIrql**. (A driver should specify a value other than PASSIVE_LEVEL only if the ISR must run above a certain IRQL.) Starting with Windows 8, a set of line-based interrupt resources that are assigned to a device can share an ISR that runs at IRQL = PASSIVE_LEVEL. If **SynchronizeIrql** = PASSIVE_LEVEL, and the maximum IRQL of this set of interrupts is PASSIVE_LEVEL, the *InterruptService* routine is called at PASSIVE_LEVEL. For more information, see Using Passive-Level Interrupt Service Routines.

FloatingSave

Specifies if the system saves the processor's floating-point state when the interrupt occurs. If **TRUE**, the system saves floating-point state. For x86-based and Itanium-based platforms, this value must be set to **FALSE**. For more information about saving floating-point and MMX state, see Using Floating Point or MMX in a WDM Driver.

`MessageBased`

Specifies the additional parameters of the operation to be performed by IoConnectInterruptEx when **Version** has a value of CONNECT_MESSAGE_BASED. For more information, see Using the CONNECT_MESSAGE_BASED Version of IoConnectInterruptEx.

PhysicalDeviceObject

A pointer to the PDO of the device.

ConnectionContext

A pointer to a location that receives a pointer to the connection context. If on return **Version** has a value of CONNECT_LINE_BASED, the routine provides a pointer to a KINTERRUPT structure. If on return **Version** has a

value of CONNECT_MESSAGE_BASED, the routine provides a pointer to an IO_INTERRUPT_MESSAGE_INFO structure.

To minimize casting, **ConnectionContext** is defined as a union. Use **ConnectionContext.Generic** to treat the location as a PVOID. Use **ConnectionContext.InterruptObject** and **ConnectionContext.InterruptMessageTable** to treat the location as a PKINTERRUPT or PIO_INTERRUPT_MESSAGE_INFO variable respectively.

`Generic`

A pointer to a PVOID variable into which the **IoConnectInterruptEx** routine writes a pointer to the connection context.

`InterruptMessageTable`

A pointer to a PIO_INTERRUPT_MESSAGE_INFO variable into which the **IoConnectInterruptEx** routine writes a pointer to the connection context.

`InterruptObject`

A pointer to a PKINTERRUPT variable into which the **IoConnectInterruptEx** routine writes a pointer to the connection context.

MessageServiceRoutine

A pointer to the InterruptMessageService routine to register as the ISR for the device's interrupts.

ServiceContext

Specifies the value to be passed as the *ServiceContext* parameter of the InterruptMessageService or InterruptService routine for the interrupt.

SpinLock

Either a pointer to a spin lock to serve as the interrupt spin lock for the set of interrupts, or **NULL**. If **NULL**, the system allocates a spin lock to serve as the interrupt spin lock. If non-**NULL**, you should have initialized the spin lock with KeInitializeSpinLock.

SynchronizeIrql

Specifies the minimum device IRQL (DIRQL) at which the ISR runs. The system only uses this value if it is greater than the maximum IRQL of the set of interrupts; otherwise, the system uses the maximum IRQL. Drivers almost always specify PASSIVE_LEVEL for **SynchronizeIrql**.

FloatingSave

Specifies if the system saves the processor's floating-point state when the interrupt occurs. If **TRUE**, the system saves floating-point state.

FallBackServiceRoutine

A pointer to an InterruptService routine to use as the ISR for line-based interrupts. If the device has no message-signaled interrupts, but has line-based interrupts, the system registers this routine to handle the line-based interrupts.

Remarks

The IoConnectInterruptEx routine takes a single *Parameters* parameter, which points to an **IO_CONNECT_INTERRUPT_PARAMETERS** structure that contains all of the parameters of the operation.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoConnectInterruptEx

IO_CONTAINER_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_CONTAINER_INFORMATION_CLASS** enumeration contains constants that indicate the classes of system information that a kernel-mode driver can request.

Syntax

```
typedef enum _IO_CONTAINER_INFORMATION_CLASS {
  IoSessionStateInformation,
  IoMaxContainerInformationClass
} IO_CONTAINER_INFORMATION_CLASS;
```

Constants

IoSessionStateInformation	Session state information. A driver uses this enumeration constant to request information about a user session.
IoMaxContainerInformationClass	Specifies the maximum value in this enumeration type.

Remarks

To request system information, a driver calls the IoGetContainerInformation routine and sets this routine's *InformationClass* parameter to an **IO_CONTAINER_NOTIFICATION_CLASS** constant (other than **IoMaxContainerInformationClass**). Currently, **IoGetContainerInformation** supports only *InformationClass* = **IoSessionStateInformation**.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IoGetContainerInformation

IO_CONTAINER_NOTIFICATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_CONTAINER_NOTIFICATION_CLASS** enumeration contains constants that indicate the classes of events for which a kernel-mode driver can register to receive notifications.

Syntax

```
typedef enum _IO_CONTAINER_NOTIFICATION_CLASS {
  IoSessionStateNotification,
  IoMaxContainerNotificationClass
} IO_CONTAINER_NOTIFICATION_CLASS;
```

Constants

IoSessionStateNotification	Session state notifications. The driver uses this enumeration constant to request notifications about changes in the state of user sessions that the driver is interested in.
IoMaxContainerNotificationClass	Specifies the maximum value in this enumeration type.

Remarks

To register for notifications, a driver calls the IoRegisterContainerNotification routine and sets this routine's *NotificationClass* parameter to an **IO_CONTAINER_NOTIFICATION_CLASS** constant (other than **IoMaxContainerNotificationClass**). Currently, **IoRegisterContainerNotification** supports only *NotificationClass* = **IoSessionStateNotification**.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IoRegisterContainerNotification

IO_DISCONNECT_INTERRUPT_PARAMETERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_DISCONNECT_INTERRUPT_PARAMETERS** structure describes the parameters when unregistering an interrupt-handling routine with IoDisconnectInterruptEx.

Syntax

```
typedef struct _IO_DISCONNECT_INTERRUPT_PARAMETERS {
  ULONG Version;
  union {
    PVOID                      Generic;
    PKINTERRUPT                InterruptObject;
    PIO_INTERRUPT_MESSAGE_INFO InterruptMessageTable;
  } ConnectionContext;
} IO_DISCONNECT_INTERRUPT_PARAMETERS, *PIO_DISCONNECT_INTERRUPT_PARAMETERS;
```

Members

`Version`

Specifies the particular operation to be performed by **IoDisconnectInterruptEx**. The value for **Version** depends on the value specified for *Parameters*->**Version** when IoConnectInterruptEx registered the interrupt handing routine, as shown in the following table.

IO_CONNECT_INTERRUPT_PARAMETERS VERSION VALUE	IO_DISCONNECT_INTERRUPT_PARAMETERS VERSION VALUE
CONNECT_LINE_BASED	CONNECT_LINE_BASED
CONNECT_MESSAGE_BASED	The value of **Version** output by **IoConnectInterruptEx**.
CONNECT_FULLY_SPECIFIED	CONNECT_FULLY_SPECIFIED

`ConnectionContext`

A pointer to the KINTERRUPT or IO_INTERRUPT_MESSAGE_INFO structure that was provided by IoConnectInterruptEx when the interrupt was connected. The value for **ConnectionContext** depends on the value specified for *Parameters*->**Version** when **IoConnectInterruptEx** registered the interrupt handling routine, as shown in the following table.

IOCONNECTINTERRUPTEX VERSION VALUE	IODISCONNECTINTERRUPTEX CONNECTIONCONTEXT VALUE
CONNECT_LINE_BASED	The value stored in **LineBased.InterruptObject**.
CONNECT_MESSAGE_BASED	The value stored in **MessageBased.ConnectionContext**.
CONNECT_FULLY_SPECIFIED	The value stored in **FullySpecified.InterruptObject**.

To minimize casting, **ConnectionContext** is defined as a union. Use **ConnectionContext.Generic** to specify the

value as a PVOID. Use **ConnectionContext.InterruptObject** and **ConnectionContext.InterruptMessageTable** to specify the value as a pointer to a KINTERRUPT or IO_INTERRUPT_MESSAGE_INFO structure.

`ConnectionContext.Generic`

A pointer to a data structure of unspecified type.

`ConnectionContext.InterruptObject`

A pointer to a KINTERRUPT structure.

`ConnectionContext.InterruptMessageTable`

A pointer to an IO_INTERRUPT_MESSAGE_INFO structure.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_INTERRUPT_MESSAGE_INFO

IoDisconnectInterruptEx

KINTERRUPT

IO_ERROR_LOG_PACKET structure

1/30/2019 • 3 minutes to read • Edit Online

The **IO_ERROR_LOG_PACKET** structure serves as the header for an error log entry.

Syntax

```
typedef struct _IO_ERROR_LOG_PACKET {
  UCHAR         MajorFunctionCode;
  UCHAR         RetryCount;
  USHORT        DumpDataSize;
  USHORT        NumberOfStrings;
  USHORT        StringOffset;
  USHORT        EventCategory;
  NTSTATUS      ErrorCode;
  ULONG         UniqueErrorValue;
  NTSTATUS      FinalStatus;
  ULONG         SequenceNumber;
  ULONG         IoControlCode;
  LARGE_INTEGER DeviceOffset;
  ULONG         DumpData[1];
} IO_ERROR_LOG_PACKET, *PIO_ERROR_LOG_PACKET;
```

Members

`MajorFunctionCode`

Indicates the **IRP_MJ_*XXX*** major function code of the IRP the driver was handling when the error occurred. Setting this value is optional.

`RetryCount`

Indicates the number of times the driver has retried the operation and encountered this error. Use zero to indicate the driver attempted the operation once, or add one for each retry beyond the initial attempt.

`DumpDataSize`

Indicates the size, in bytes, of the variable-length **DumpData** member of this structure. The specified value must be a multiple of **sizeof**(ULONG).

`NumberOfStrings`

Indicates the number of insertion strings the driver will supply with this error log entry. Drivers set this value to zero for errors that need no insertion strings. The Event Viewer uses these strings to fill in the "%2" through "%*n*" entries in the string template for this error code.

The null-terminated Unicode strings themselves follow the **IO_ERROR_LOG_PACKET** structure in memory.

`StringOffset`

Indicates the offset, in bytes, from the beginning of the structure, at which any driver-supplied insertion string data begins. Normally this will be **sizeof**(**IO_ERROR_LOG_PACKET**) plus the value of the **DumpDataSize** member. If there are no driver-supplied insertion strings, **StringOffset** can be zero.

`EventCategory`

Specifies the event category for the error. A driver specifies the event categories it supports and corresponding descriptive strings in its message catalog. The Event Viewer displays the descriptive string as the **Category** value for the error.

`ErrorCode`

Specifies the type of error. The Event Viewer uses the error code to determine which string to display as the Description value for the error. The Event Viewer takes the string template for the error supplied in the driver's message catalog, replaces "%1" in the template with the name of the driver's device object, and replaces "%2" through "%*n*" with the insertion strings supplied with the error log entry.

ErrorCode is a system-defined or driver-defined constant; see Logging Errors for more information.

`UniqueErrorValue`

A driver-specific value that indicates where the error was detected in the driver. Setting this value is optional.

`FinalStatus`

Specifies the NTSTATUS value to be returned for the operation that triggered the error. Setting this value is optional.

`SequenceNumber`

Specifies a driver-assigned sequence number for the current IRP, which should be constant for the life of a given request. Setting this value is optional.

`IoControlCode`

For an IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL IRP, this member specifies the I/O control code for the request that trigged the error. Otherwise, this value is zero. Setting this value is optional.

`DeviceOffset`

Specifies the driver-specified offset into the device where the error occurred. Setting this value is optional.

`DumpData`

A variable-size array that can be used to store driver-specific binary data, such as register values or any other information useful in identifying the cause of the error. Drivers must specify the size, in bytes, of the array in the **DumpDataSize** member of this structure.

Remarks

Drivers use the IoAllocateErrorLogEntry routine to allocate an error log entry. The **IO_ERROR_LOG_PACKET** structure serves as the header for the returned buffer. It is followed in memory by any insertion strings for the log entry.

Note that the I/O manager itself inserts some information into the system error log, such as the name of the device and driver. The I/O manager reserves 80 bytes to hold this information. If the size of this information exceeds 80 bytes, then the I/O manager truncates the driver's insertion strings as necessary.

For more information about how to use this structure, see Logging Errors.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IRP_MJ_DEVICE_CONTROL

IRP_MJ_INTERNAL_DEVICE_CONTROL

IoAllocateErrorLogEntry

IoFreeErrorLogEntry

IoWriteErrorLogEntry

IO_INTERRUPT_MESSAGE_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_INTERRUPT_MESSAGE_INFO** structure describes the driver's message-signaled interrupts.

Syntax

```
typedef struct _IO_INTERRUPT_MESSAGE_INFO {
  KIRQL                           UnifiedIrql;
  ULONG                           MessageCount;
  IO_INTERRUPT_MESSAGE_INFO_ENTRY MessageInfo[1];
} IO_INTERRUPT_MESSAGE_INFO, *PIO_INTERRUPT_MESSAGE_INFO;
```

Members

`UnifiedIrql`

Specifies the IRQL at which the driver's InterruptMessageService routine runs. If the driver specified an interrupt spin lock for its interrupts, **UnifiedIrql** also specifies the IRQL at which the interrupt spin lock is acquired.

`MessageCount`

Specifies the number of message-signaled interrupts allocated for the driver.

`MessageInfo`

Variable-length array that specifies an IO_INTERRUPT_MESSAGE_INFO_ENTRY structure for each message-signaled interrupt allocated for the driver. The **MessageCount** member specifies the number of entries in this array.

Remarks

When a driver calls IoConnectInterruptEx to register an InterruptMessageService routine, **IoConnectInterruptEx** provides a pointer to **IO_INTERRUPT_MESSAGE_INFO** in *Parameters*->**MessageBased.ConnectionContext**. For more information, see IO_CONNECT_INTERRUPT_PARAMETERS.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_CONNECT_INTERRUPT_PARAMETERS

IO_INTERRUPT_MESSAGE_INFO_ENTRY

IoConnectInterruptEx

IO_INTERRUPT_MESSAGE_INFO_ENTRY structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_INTERRUPT_MESSAGE_INFO_ENTRY** structure describes the properties of a single message-signaled interrupt.

Syntax

```
typedef struct _IO_INTERRUPT_MESSAGE_INFO_ENTRY {
  PHYSICAL_ADDRESS    MessageAddress;
  KAFFINITY           TargetProcessorSet;
  PKINTERRUPT         InterruptObject;
  ULONG               MessageData;
  ULONG               Vector;
  KIRQL               Irql;
  KINTERRUPT_MODE     Mode;
  KINTERRUPT_POLARITY Polarity;
} IO_INTERRUPT_MESSAGE_INFO_ENTRY, *PIO_INTERRUPT_MESSAGE_INFO_ENTRY;
```

Members

`MessageAddress`

Specifies the physical address that triggers the interrupt message.

`TargetProcessorSet`

Specifies a **KAFFINITY** value that determines the set of processors that can receive the interrupt.

`InterruptObject`

Pointer to the interrupt object that represents the interrupt.

`MessageData`

Specifies the value to be written to the address specified by **MessageAddress** to trigger the interrupt.

`Vector`

Specifies the interrupt vector for the interrupt.

`Irql`

Specifies the device IRQL (DIRQL) for the interrupt.

`Mode`

Specifies a KINTERRUPT_MODE value that determines whether the interrupt is level-sensitive or latched.

`Polarity`

Specifies a KINTERRUPT_POLARITY value that determines whether the interrupt is active-high or active-low.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_INTERRUPT_MESSAGE_INFO

IO_PAGING_PRIORITY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_PAGING_PRIORITY** enumeration describes the priority value for a paging I/O IRP.

Syntax

```
typedef enum _IO_PAGING_PRIORITY {
  IoPagingPriorityInvalid,
  IoPagingPriorityNormal,
  IoPagingPriorityHigh,
  IoPagingPriorityReserved1,
  IoPagingPriorityReserved2
} IO_PAGING_PRIORITY;
```

Constants

IoPagingPriorityInvalid	The IRP is not a paging I/O IRP.
IoPagingPriorityNormal	The associated IRP has a normal priority level for paging I/O.
IoPagingPriorityHigh	The associated IRP has a high priority level for paging I/O.
IoPagingPriorityReserved1	Reserved for system use.
IoPagingPriorityReserved2	Reserved for system use.

Remarks

The IoGetPagingIoPriority routine returns an **IO_PAGING_PRIORITY** value to indicate the priority value of a paging I/O IRP.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoGetPagingIoPriority

IO_PRIORITY_HINT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_PRIORITY_HINT** enumeration type specifies the priority hint for an IRP.

Syntax

```
typedef enum _IO_PRIORITY_HINT {
  IoPriorityVeryLow,
  IoPriorityLow,
  IoPriorityNormal,
  IoPriorityHigh,
  IoPriorityCritical,
  MaxIoPriorityTypes
} IO_PRIORITY_HINT;
```

Constants

IoPriorityVeryLow	Specifies the lowest possible priority hint level. The system uses this value for background I/O operations.
IoPriorityLow	Specifies a low-priority hint level.
IoPriorityNormal	Specifies a normal-priority hint level. This value is the default setting for an IRP.
IoPriorityHigh	Specifies a high-priority hint level. This value is reserved for use by the system.
IoPriorityCritical	Specifies the highest-priority hint level. This value is reserved for use by the system.
MaxIoPriorityTypes	Marks the limit for priority hints. Any priority hint value must be less than **MaxIoPriorityTypes**.

Remarks

For more information about priority hints, see Using IRP Priority Hints.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoGetIoPriorityHint

IoSetIoPriorityHint

IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure contains the connection context for a registered interrupt service routine (ISR) that was connected to an interrupt or interrupts by a previous call to the IoConnectInterruptEx routine.

Syntax

```
typedef struct _IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS {
  ULONG Version;
  union {
    PVOID                      Generic;
    PKINTERRUPT                InterruptObject;
    PIO_INTERRUPT_MESSAGE_INFO InterruptMessageTable;
  } ConnectionContext;
} IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS, *PIO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS;
```

Members

`Version`

The interrupt connection type described by this structure. This member must match the connection type the caller obtained from the **IoConnectInterruptEx** call that connected the interrupt. The following are valid values for this member:

- CONNECT_FULLY_SPECIFIED
- CONNECT_LINE_BASED
- CONNECT_MESSAGE_BASED
- CONNECT_FULLY_SPECIFIED_GROUP

For more information, see Remarks.

`ConnectionContext`

A union that contains the connection context associated with the interrupt. Each union member is a pointer to a connection context of a particular type. If **Version** = CONNECT_MESSAGE_BASED, the **InterruptMessageTable** member of this union contains a pointer to an IO_INTERRUPT_MESSAGE_INFO structure. For all other **Version** values, the **InterruptObject** member of this union contains a pointer to a KINTERRUPT structure. Use the **Generic** member of this union to treat the connection context pointer as a PVOID value (and avoid casting).

For more information, see Remarks.

`ConnectionContext.Generic`

A PVOID pointer to a connection context.

`ConnectionContext.InterruptObject`

A pointer to a connection context of type KINTERRUPT.

`ConnectionContext.InterruptMessageTable`

A pointer to a connection context of type IO_INTERRUPT_MESSAGE_INFO.

Remarks

This structure is supplied as a parameter to the IoReportInterruptActive and IoReportInterruptInactive routines.

The **Version** member indicates the type of the connection context specified in the **ConnectionContext** member. Set this member to the *Parameters*->**Version** value obtained from the IoConnectInterruptEx call that connected the interrupt.

If the *Parameters*->**Version** value obtained from **IoConnectInterruptEx** is CONNECT_FULLY_SPECIFIED or CONNECT_FULLY_SPECIFIED_GROUP, set the **ConnectionContext.InterruptObject** member of the **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure to the *(*Parameters*->**FullySpecified.InterruptObject**) pointer value obtained from **IoConnectInterruptEx**.

If the *Parameters*->**Version** value obtained from **IoConnectInterruptEx** is CONNECT_MESSAGE_BASED, set the **ConnectionContext.InterruptMessageTable** member of the **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure to the *(*Parameters*->**MessageBased.ConnectionContext.InterruptMessageTable**) pointer value obtained from **IoConnectInterruptEx**.

If the *Parameters*->**Version** value obtained from **IoConnectInterruptEx** is CONNECT_LINE_BASED, but the caller initially set *Parameters*->**Version** to CONNECT_MESSAGE_BASED, set the **ConnectionContext.InterruptObject** member of the **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure to the *(*Parameters*->**MessageBased.ConnectionContext.InterruptObject**) pointer value obtained from **IoConnectInterruptEx**.

If the *Parameters*->**Version** value obtained from **IoConnectInterruptEx** is CONNECT_LINE_BASED, and the caller initially set *Parameters*->**Version** to CONNECT_LINE_BASED, set the **ConnectionContext.InterruptObject** member of the **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure to the *(*Parameters*->**LineBased.InterruptObject**) pointer value obtained from **IoConnectInterruptEx**.

Requirements

Header	wdm.h (include Wdm.h)

See Also

IoConnectInterruptEx

IoReportInterruptActive

IoReportInterruptInactive

IO_RESOURCE_DESCRIPTOR structure

1/30/2019 • 7 minutes to read • Edit Online

The **IO_RESOURCE_DESCRIPTOR** structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of **IO_RESOURCE_DESCRIPTOR** structures is contained within each IO_RESOURCE_LIST structure.

Syntax

```
typedef struct _IO_RESOURCE_DESCRIPTOR {
  UCHAR  Option;
  UCHAR  Type;
  UCHAR  ShareDisposition;
  UCHAR  Spare1;
  USHORT Flags;
  USHORT Spare2;
  union {
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Port;
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory;
    struct {
      ULONG             MinimumVector;
      ULONG             MaximumVector;
      IRQ_DEVICE_POLICY AffinityPolicy;
      USHORT            Group;
      IRQ_PRIORITY      PriorityPolicy;
      KAFFINITY         TargetedProcessors;
    } Interrupt;
    struct {
      ULONG MinimumChannel;
      ULONG MaximumChannel;
    } Dma;
    struct {
      ULONG RequestLine;
      ULONG Reserved;
      ULONG Channel;
      ULONG TransferWidth;
    } DmaV3;
    struct {
      ULONG            Length;
      ULONG            Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Generic;
    struct {
      ULONG Data[3];
    } DevicePrivate;
    struct {
      ULONG Length;
      ULONG MinBusNumber;
      ULONG MaxBusNumber;
      ULONG Reserved;
```

```
      ULONG Reserved;
    } BusNumber;
    struct {
      ULONG Priority;
      ULONG Reserved1;
      ULONG Reserved2;
    } ConfigData;
    struct {
      ULONG            Length40;
      ULONG            Alignment40;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory40;
    struct {
      ULONG            Length48;
      ULONG            Alignment48;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory48;
    struct {
      ULONG            Length64;
      ULONG            Alignment64;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory64;
    struct {
      UCHAR Class;
      UCHAR Type;
      UCHAR Reserved1;
      UCHAR Reserved2;
      ULONG IdLowPart;
      ULONG IdHighPart;
    } Connection;
  } u;
} IO_RESOURCE_DESCRIPTOR, *PIO_RESOURCE_DESCRIPTOR;
```

Members

`Option`

Specifies whether this resource description is required, preferred, or alternative. One of the following values must be used:

VALUE	MEANING
0	The specified resource range is required, unless alternative ranges are also specified.
IO_RESOURCE_PREFERRED	The specified resource range is preferred to any alternative ranges.

IO_RESOURCE_ALTERNATIVE	The specified resource range is an alternative to the range preceding it. For example, if one **IO_RESOURCE_DESCRIPTOR** structure specifies IRQ 5, with IO_RESOURCE_PREFERRED set, and the next structure specifies IRQ 3, with IO_RESOURCE_ALTERNATIVE set, the PnP manager assigns IRQ 3 to the device only if IRQ 5 is unavailable. (Multiple alternatives can be specified for each resource. Both IO_RESOURCE_ALTERNATIVE and IO_RESOURCE_PREFERRED can be set, indicating a preferred alternative.)
IO_RESOURCE_DEFAULT	Not used.

`Type`

Identifies the resource type. For a list of valid values, see the **Type** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`ShareDisposition`

Indicates whether the described resource can be shared. For a list of valid values, see the **ShareDisposition** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`Spare1`

`Flags`

Contains bit flags that are specific to the resource type. The following table shows the flags that are valid if **Type** = **CmResourceTypeInterrupt.**

VALUE	MEANING
CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE	The IRQ line is level-triggered. (These IRQs are usually sharable.)

CM_RESOURCE_INTERRUPT_LATCHED	The IRQ line is edge-triggered.
CM_RESOURCE_INTERRUPT_MESSAGE	If this flag is set, the interrupt is a message-signaled interrupt. Otherwise, the interrupt is a line-based interrupt. This flag can be set starting with Windows Vista.
CM_RESOURCE_INTERRUPT_POLICY_INCLUDED	If this flag is set, the u.Interrupt member includes data that describes the device's interrupt policy. This flag can be set starting with Windows Vista.
CM_RESOURCE_INTERRUPT_SECONDARY_INTERRUPT	The interrupt is a secondary interrupt. This flag can be set starting with Windows 8. For more information about secondary interrupts, see GPIO Interrupts.

CM_ RES OU RCE _IN TER RUP T_W AKE _HI NT	The interrupt is capable of waking the operating system from a low-power idle state or a system sleep state. This flag can be set starting with Windows 8. For more information about wake capabilities, see Enabling Device Wake-Up.

For a list of valid flags for other resource types, see the description of the **Flags** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

`Spare2`

`u`

port

Specifies a range of I/O port addresses, using the following members.

Drivers for Windows Vista and later versions of the Windows operating system must use RtlIoDecodeMemIoResource and RtlIoEncodeMemIoResource to read and update this member, rather than updating it directly.

Length

The length, in bytes, of the range of assignable I/O port addresses.

Alignment

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be an integer multiple of *Alignment*.

MinimumAddress

The minimum bus-relative I/O port address that can be assigned to the device.

MaximumAddress

The maximum bus-relative I/O port address that can be assigned to the device.

`u.Port`

`u.Port.Length`

`u.Port.Alignment`

`u.Port.MinimumAddress`

`u.Port.MaximumAddress`

`u.Memory`

`u.Memory.Length`

The length, in bytes, of the range of assignable memory addresses.

`u.Memory.Alignment`

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be an integer multiple of *Alignment*.

`u.Memory.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Interrupt`

`u.Interrupt.MinimumVector`

The minimum bus-relative vector that can be assigned to the device.

`u.Interrupt.MaximumVector`

The maximum bus-relative vector that can be assigned to the device.

If the **CM_RESOURCE_INTERRUPT_MESSAGE** flag bit is set, the values of the **MinimumVector** and **MaximumVector** members have special meanings. For more information, see Using Interrupt Resource Descriptors.

The system supplies the following additional members on Windows Vista and later versions of the Windows operating system:

`u.Interrupt.AffinityPolicy`

Specifies an IRQ_DEVICE_POLICY value that indicates how the system should distribute a device's interrupts between processors.

Specifies an IRQ_DEVICE_POLICY value that indicates how the system should distribute a device's interrupts between processors.

`u.Interrupt.Group`

Specifies a processor group number. **Group** is a valid (but optional) member of **u.Interrupt** only in Windows 7 and later versions of Windows. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. If the **Group** member exists, the **Group** and **TargetedProcessors** members together specify a group affinity that identifies the set of processors that should handle the device's interrupts. To specify an affinity for a particular group, set **AffinityPolicy** to **IrqPolicySpecifiedProcessors** and set **Group** to the appropriate group number. In addition, **TargetedProcessors** must specify the target processors in the group. If you set **AffinityPolicy** to a value other than **IrqPolicySpecifiedProcessors**, set **Group** to ALL_PROCESSOR_GROUPS to indicate that the driver is group-aware (that is, designed to handle information about processor groups). A driver cannot specify target processors if **Group** equals ALL_PROCESSOR_GROUPS; such target specifications are ignored.

`u.Interrupt.PriorityPolicy`

Specifies an IRQ_PRIORITY value that indicates the priority with which the system should dispatch the device's interrupts.

`u.Interrupt.TargetedProcessors`

Specifies a **KAFFINITY** value that indicates which processors should handle the device's interrupts. This value is used only if **AffinityPolicy** is **IrqPolicySpecifiedProcessors.**

`u.Dma`

`u.Dma.MinimumChannel`

The minimum bus-relative DMA channel that can be assigned to the device.

`u.Dma.MaximumChannel`

The maximum bus-relative DMA channel that can be assigned to the device.

`u.DmaV3`

`u.DmaV3.RequestLine`

The number of the request line on the system DMA controller that is allocated to the device.

`u.DmaV3.Reserved`

Not used.

`u.DmaV3.Channel`

The number of the DMA channel on the system DMA controller that is allocated to the device.

`u.DmaV3.TransferWidth`

Specifies the width, in bits, of the data bus that the system DMA controller that is allocated to the device uses to transfer data to or from the device.

`u.Generic`

`u.Generic.Length`

`u.Generic.Alignment`

`u.Generic.MinimumAddress`

`u.Generic.MaximumAddress`

`u.DevicePrivate`

`u.DevicePrivate.Data`

`u.BusNumber`

`u.BusNumber.Length`

The number of bus numbers required.

`u.BusNumber.MinBusNumber`

The minimum bus-relative bus number that can be assigned to the device.

`u.BusNumber.MaxBusNumber`

The maximum bus-relative bus number that can be assigned to the device.

`u.BusNumber.Reserved`

Not used.

`u.ConfigData`

`u.ConfigData.Priority`

`u.ConfigData.Reserved1`

`u.ConfigData.Reserved2`

`u.Memory40`

`u.Memory40.Length40`

The high 32 bits of the 40-bit length, in bytes, of the range of assignable memory addresses. The lower 8 bits are treated as zero.

`u.Memory40.Alignment40`

The high 32 bits of the 40-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 8 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory40.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory40.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Memory48`

`u.Memory48.Length48`

The high 32 bits of the 48-bit length, in bytes, of the range of assignable memory addresses. The lower 16 bits are treated as zero.

`u.Memory48.Alignment48`

The high 32 bits of the 48-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 16 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory48.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory48.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Memory64`

`u.Memory64.Length64`

The high 32 bits of the 64-bit length, in bytes, of the range of assignable memory addresses. The lower 32 bits are treated as zero.

`u.Memory64.Alignment64`

The high 32 bits of the 64-bit alignment, in bytes, that the assigned starting address must adhere to. The lower 32 bits are treated as zero. The assigned starting address will be a multiple of the alignment.

`u.Memory64.MinimumAddress`

The minimum bus-relative memory address that can be assigned to the device.

`u.Memory64.MaximumAddress`

The maximum bus-relative memory address that can be assigned to the device.

`u.Connection`

`u.Connection.Class`

The connection class. This member is set to one of the following values.

VALUE	MEANING

CM_RESOURCE_CONNECTION_CLASS_GPIO	Access the device through one or more pins on a GPIO controller.
CM_RESOURCE_CONNECTION_CLASS_SERIAL	Access the device through a serial bus or serial port.

`u.Connection.Type`

The connection type. If *Class* = CM_RESOURCE_CONNECTION_CLASS_GPIO, *Type* is set to the following value:

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_GPIO_IO	Access the device through GPIO pins that are configured for I/O.

Note A GPIO pin that is configured as an interrupt request input is accessed as an ordinary interrupt resource (**CmResourceTypeInterrupt**).

If **Class** = CM_RESOURCE_CONNECTION_CLASS_SERIAL, **Type** is set to one of the following values:

VALUE	MEANING

CM_RESOURCE_CONNECTION_TYPE_I2C	The device is connected to an I²C bus.
CM_RESOURCE_CONNECTION_TYPE_SPI	The device is connected to an SPI bus.
CM_RESOURCE_CONNECTION_TYPE_SERIAL_UART	The device is connected to a serial port.

`u.Connection.Reserved1`

Not used.

`u.Connection.Reserved2`

Not used.

`u.Connection.IdLowPart`

The lower 32 bits of the 64-bit connection ID.

`u.Connection.IdHighPart`

The upper 32 bits of the 64-bit connection ID.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IO_RESOURCE_LIST

IO_RESOURCE_REQUIREMENTS_LIST

IoConnectInterrupt

IO_RESOURCE_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_RESOURCE_LIST** structure describes a range of raw hardware resources, of various types, that can be used by a device. The resources specified represent a single, acceptable resource configuration for a device. An array of **IO_RESOURCE_LIST** structures is contained within each IO_RESOURCE_REQUIREMENTS_LIST structure.

Syntax

```
typedef struct _IO_RESOURCE_LIST {
  USHORT                  Version;
  USHORT                  Revision;
  ULONG                   Count;
  IO_RESOURCE_DESCRIPTOR Descriptors[1];
} IO_RESOURCE_LIST, *PIO_RESOURCE_LIST;
```

Members

`Version`

The version number of this structure. This value should be 1.

`Revision`

The revision of this structure. This value should be 1.

`Count`

The number of elements in the **Descriptors** array.

`Descriptors`

The first element in an array of one or more IO_RESOURCE_DESCRIPTOR structures.

Remarks

This structure is the header for an array of **IO_RESOURCE_DESCRIPTOR** structures. The **Descriptors** member contains the first element in this array, and the **Count** member specifies the total number of array elements. If the array contains more than one element, the remaining elements in the array immediately follow the **IO_RESOURCE_LIST** structure in memory. The total number of bytes occupied by the **IO_RESOURCE_LIST** structure and any array elements that follow this structure is **sizeof**(**IO_RESOURCE_LIST**) + (**Count** - 1) * **sizeof**(**IO_RESOURCE_DESCRIPTOR**).

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IO_RESOURCE_REQUIREMENTS_LIST

IO_RESOURCE_REQUIREMENTS_LIST structure

1/30/2019 • 3 minutes to read • Edit Online

The **IO_RESOURCE_REQUIREMENTS_LIST** structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.

Syntax

```
typedef struct _IO_RESOURCE_REQUIREMENTS_LIST {
  ULONG            ListSize;
  INTERFACE_TYPE   InterfaceType;
  ULONG            BusNumber;
  ULONG            SlotNumber;
  ULONG            Reserved[3];
  ULONG            AlternativeLists;
  IO_RESOURCE_LIST List[1];
} IO_RESOURCE_REQUIREMENTS_LIST, *PIO_RESOURCE_REQUIREMENTS_LIST;
```

Members

`ListSize`

The total number of bytes that constitute the **IO_RESOURCE_REQUIREMENTS_LIST** structure, its IO_RESOURCE_LIST array, and the latter's IO_RESOURCE_DESCRIPTOR array.

`InterfaceType`

Specifies an interface type. This must be one of the types defined by INTERFACE_TYPE, in Wdm.h or Ntddk.h. (Not used by WDM drivers.)

`BusNumber`

A system-assigned, zero-based bus number. (Not used by WDM drivers.)

`SlotNumber`

A system slot number. (Not used by WDM drivers.)

`Reserved`

Not used.

`AlternativeLists`

The number of alternative lists (or configurations) of hardware resources that are described by this **IO_RESOURCE_REQUIREMENTS_LIST** structure. The **List** member is the header for the first alternative list.

`List`

The IO_RESOURCE_LIST structure that serves as the header for the first alternative list of hardware resources. If the **IO_RESOURCE_REQUIREMENTS_LIST** structure describes more than one alternative list, the second alternative list immediately follows the first alternative list in memory, and so on. The size of each alternative list depends on the length of the IO_RESOURCE_DESCRIPTOR array that it contains. For more information, see the following Remarks section.

Remarks

This structure is used by the IRP_MN_FILTER_RESOURCE_REQUIREMENTS and IRP_MN_QUERY_RESOURCE_REQUIREMENTS IRPs to describe one or more alternative configurations of hardware resources.

The **IO_RESOURCE_REQUIREMENTS_LIST** structure is a header for a larger data structure, of variable size, that specifies one or more alternative lists of resources. All of the data in this larger structure occupies a contiguous block of memory, whose size is specified by the **ListSize** member. Each alternative list occupies a subblock within the larger block.

Each alternative list begins with an **IO_RESOURCE_LIST** structure, which serves as a header for an array of **IO_RESOURCE_DESCRIPTOR** structures. The length of this array determines the size of the list, which can vary from one alternative list to the next. Driver code can use pointer arithmetic to step from one alternative list to the next. For example, if a parameter named *list* is a pointer to the **IO_RESOURCE_LIST** structure at the start of one alternative list, *list* can be updated to point to the start of the next list as follows:

```
list = (PIO_RESOURCE_LIST)(list->Descriptors + list->Count);
```

In this example, `list->Descriptors` is a pointer to the start of the **IO_RESOURCE_DESCRIPTOR** array, and `list->Count` is the number of elements in the array. For more information about the **Descriptors** and **Count** members, see IO_RESOURCE_LIST. #### Examples If a driver contains a handler for an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** IRP, the handler typically walks the resource list that is pointed to by the **IoStatus.Information** field in the IRP. The following code example contains a function—named FilterResources—that is called in the handler to walk the list. This function either verifies that a particular resource is specified in the list, or changes the resource requirements in the list without changing the size of the list. (A function that changes the size of the list would need to allocate a buffer for the updated list, write the list to this buffer, update the **IoStatus.Information** field in the IRP to point to the updated list, and free the buffer that contains the original list.) The FilterResources function returns **TRUE** (probably from the **switch** statement, although the details are omitted to simplify the code example) if it finds the resource that it is looking for in the list, or if it makes a change to the list that does not change the size of the list. Otherwise, it returns **FALSE**.

```
/* Modify the resource list in place (without changing its size). */

BOOLEAN FilterResources(PIO_RESOURCE_REQUIREMENTS_LIST reqlist)
{
    PIO_RESOURCE_LIST list;

    list = reqlist->List;

    for (int ix = 0; ix < reqlist->AlternativeLists; ++ix)
    {
        /* Process resources in IO_RESOURCE_LIST block number ix. */

        for (int jx = 0; jx < list->Count; ++jx)
        {
            PIO_RESOURCE_DESCRIPTOR desc;

            desc = list->Descriptors + jx;

            switch (desc->Type)
            {
               /* Process element jx in Descriptors array. */

               ...

            }
        }

        /* Advance to next IO_RESOURCE_LIST block in memory. */

        list = (PIO_RESOURCE_LIST)(list->Descriptors + list->Count);
    }

    return FALSE;
}
```

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IO_RESOURCE_LIST

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

IRP_MN_QUERY_RESOURCE_REQUIREMENTS

IO_SECURITY_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_SECURITY_CONTEXT** structure represents the security context of an IRP_MJ_CREATE request.

Syntax

```
typedef struct _IO_SECURITY_CONTEXT {
  PSECURITY_QUALITY_OF_SERVICE SecurityQos;
  PACCESS_STATE                AccessState;
  ACCESS_MASK                  DesiredAccess;
  ULONG                        FullCreateOptions;
} IO_SECURITY_CONTEXT, *PIO_SECURITY_CONTEXT;
```

Members

`SecurityQos`

Reserved for system use.

`AccessState`

Reserved for use by file systems and file system filter drivers. This member is a pointer to an ACCESS_STATE structure that contains the object's subject context, granted access types, and remaining desired access types.

`DesiredAccess`

An ACCESS_MASK value that expresses the access rights that are requested in the IRP_MJ_CREATE request.

`FullCreateOptions`

Reserved for system use.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

ACCESS_STATE

IRP_MJ_CREATE

IO_SESSION_CONNECT_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_SESSION_CONNECT_INFO** structure provides information about a user session.

Syntax

```
typedef struct _IO_SESSION_CONNECT_INFO {
  ULONG   SessionId;
  BOOLEAN LocalSession;
} IO_SESSION_CONNECT_INFO, *PIO_SESSION_CONNECT_INFO;
```

Members

`SessionId`

Session ID. This member contains the Terminal Services session identifier of the user session for which the driver is receiving this notification.

`LocalSession`

Indicates whether the user session is a local session or a remote session. If **TRUE**, the user is logged on locally. If **FALSE**, the user is logged on remotely.

Remarks

If a driver is registered to receive notifications of events in a user session, and if this session enters the *connected* state, the I/O manager calls the driver's IO_SESSION_NOTIFICATION_FUNCTION function. For this call, the I/O manager sets the function's *Event* parameter to **IoSessionEventConnected**. Additionally, the I/O manager sets the function's *NotificationPayload* parameter to point to an **IO_SESSION_CONNECT_INFO** structure that contains information about the user session. For more information about **IoSessionEventConnected**, see IO_SESSION_EVENT.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IO_SESSION_EVENT

IO_SESSION_NOTIFICATION_FUNCTION

IO_SESSION_EVENT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_SESSION_EVENT** enumeration indicates the type of session event for which a driver is receiving notification.

Syntax

```
typedef enum _IO_SESSION_EVENT {
  IoSessionEventIgnore,
  IoSessionEventCreated,
  IoSessionEventTerminated,
  IoSessionEventConnected,
  IoSessionEventDisconnected,
  IoSessionEventLogon,
  IoSessionEventLogoff,
  IoSessionEventMax
} IO_SESSION_EVENT, *PIO_SESSION_EVENT;
```

Constants

IoSessionEventIgnore	
IoSessionEventCreated	The user session was created.
IoSessionEventTerminated	The user session terminated.
IoSessionEventConnected	The user session was connected.
IoSessionEventDisconnected	The user session was disconnected.
IoSessionEventLogon	The user logged on to the session.
IoSessionEventLogoff	The user logged off of the session.
IoSessionEventMax	Specifies the maximum value in this enumeration type.

Remarks

When the I/O manager calls the driver's IO_SESSION_NOTIFICATION_FUNCTION function, it sets the *Event* parameter of this function to an **IO_SESSION_EVENT** enumeration constant (other than **IoSessionEventMax**).

A session event causes a transition from one session state to another. For more information about session state transitions, see IO_SESSION_STATE.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IO_SESSION_NOTIFICATION_FUNCTION

IO_SESSION_STATE

IO_SESSION_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IO_SESSION_STATE** enumeration contains constants that indicate the current state of a user session.

Syntax

```
typedef enum _IO_SESSION_STATE {
  IoSessionStateCreated,
  IoSessionStateInitialized,
  IoSessionStateConnected,
  IoSessionStateDisconnected,
  IoSessionStateDisconnectedLoggedOn,
  IoSessionStateLoggedOn,
  IoSessionStateLoggedOff,
  IoSessionStateTerminated,
  IoSessionStateMax
} IO_SESSION_STATE, *PIO_SESSION_STATE;
```

Constants

IoSessionStateCreated	The session has been created.
IoSessionStateInitialized	The session has been initialized but has not yet been created.
IoSessionStateConnected	The session is connected but the user has not yet logged on.
IoSessionStateDisconnected	The session has been disconnected.
IoSessionStateDisconnectedLoggedOn	The session was disconnected while the user was logged on.
IoSessionStateLoggedOn	The user is logged on to the session.
IoSessionStateLoggedOff	The user has logged off of the session.
IoSessionStateTerminated	The session has been terminated.
IoSessionStateMax	Specifies the maximum value in this enumeration type.

Remarks

When a driver calls the IoGetContainerInformation routine to obtain information about a user session (*InformationClass* = **IoSessionStateInformation**), the I/O manager writes an IO_SESSION_STATE_INFORMATION structure to the buffer pointed to by the routine's *Buffer* parameter. The I/O manager sets the **SessionState** member of this structure to an **IO_SESSION_STATE** enumeration constant (other than **IoSessionStateMax**).

The following table shows the session state transitions. For each state transition, the table shows the following:

- The *from* state (a column label in a gray box)

- The *to* state (a row label in a gray box)
- The event that causes the transition (a table entry in a white box)

A blank white box indicates that no transition can occur directly from the associated *from* state to the corresponding *to* state.

		From state						
		Initialized	Created	Connected	Disconnected	Disconnected + LoggedOn	LoggedOn	LoggedOff
To state	Created	*Created*						
	Connected		*Connected*		*Connected*			
	Disconnected		*Disconnected*	*Disconnected*		*Logoff*		*Disconnected*
	Disconnected + LoggedOn				*Logon*		*Disconnected*	
	LoggedOn			*Logon*				
	LoggedOff						*Logoff*	
	Terminated		*Terminated*	*Terminated*	*Terminated*	*Terminated*	*Terminated*	*Terminated*

In the preceding table, the *from* and *to* states are represented by **IO_SESSION_STATE** enumeration constants, and the events are represented by IO_SESSION_EVENT enumeration constants. For example, if the session state is **IoSessionStateConnected** (abbreviated as "Connected" in the table), an **IoSessionEventLogon** event (abbreviated as "Logon") causes a transition to the **IoSessionStateLoggedOn** state (abbreviated as "LoggedOn"). The starting state for a new session is **IoSessionStateInitialized** (abbreviated as "Initialized").

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IO_SESSION_EVENT

IO_SESSION_STATE_INFORMATION

IoGetContainerInformation

IO_SESSION_STATE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **IO_SESSION_STATE_INFORMATION** structure contains information about the state of a user session.

Syntax

```
typedef struct _IO_SESSION_STATE_INFORMATION {
  ULONG            SessionId;
  IO_SESSION_STATE SessionState;
  BOOLEAN          LocalSession;
} IO_SESSION_STATE_INFORMATION, *PIO_SESSION_STATE_INFORMATION;
```

Members

`SessionId`

The session ID. This member contains the Terminal Services session identifier of a user session. The IoGetContainerInformation routine sets this member to the session ID of the session that is represented by the session object that the *ContainerObject* parameter of **IoGetContainerInformation** points to.

`SessionState`

The current state of the user session that is identified by *SessionId*. This member is set to one of the following IO_SESSION_STATE enumeration constants:

- **IoSessionStateCreated**
- **IoSessionStateInitialized**
- **IoSessionStateConnected**
- **IoSessionStateDisconnected**
- **IoSessionStateDisconnectedLoggedOn**
- **IoSessionStateLoggedOn**
- **IoSessionStateLoggedOff**
- **IoSessionStateTerminated**

`LocalSession`

Indicates whether the user session identified by *SessionId* is a local session. If **TRUE**, the user is logged on locally. If **FALSE**, the user is logged on remotely. This member is valid only if the session is connected. The following **IO_SESSION_STATE** enumeration constants represent session states in which the session is connected:

- **IoSessionStateConnected**
- **IoSessionStateLoggedOn**
- **IoSessionStateLoggedOff**

Remarks

To obtain information about a user session, a driver calls the **IoGetContainerInformation** routine. This routine's *Buffer* parameter points to an **IO_SESSION_STATE_INFORMATION** structure that contains information about the session state.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

IO_SESSION_STATE

IoGetContainerInformation

IO_SESSION_STATE_NOTIFICATION structure

1/30/2019 • 3 minutes to read • Edit Online

The **IO_SESSION_STATE_NOTIFICATION** structure contains information that a kernel-mode driver supplies to the IoRegisterContainerNotification routine when the driver registers to receive notifications of session events.

Syntax

```
typedef struct _IO_SESSION_STATE_NOTIFICATION {
  ULONG Size;
  ULONG Flags;
  PVOID IoObject;
  ULONG EventMask;
  PVOID Context;
} IO_SESSION_STATE_NOTIFICATION, *PIO_SESSION_STATE_NOTIFICATION;
```

Members

`Size`

The size, in bytes, of the **IO_SESSION_STATE_NOTIFICATION** structure.

`Flags`

No flags are currently defined for this member. Set to zero.

`IoObject`

A pointer to an I/O object owned by the driver. This member can point to a DEVICE_OBJECT, DRIVER_OBJECT, or FILE_OBJECT structure. The I/O object must remain valid for the lifetime of the registration. Before you delete a registered device object, unload a registered driver, or close a registered file object, call the IoUnregisterContainerNotification routine to cancel the registration. A driver can maintain simultaneous registrations for more than one I/O object, but it cannot create more than one active registration for the same I/O object.

`EventMask`

Mask bits for session events. These mask bits indicate the events for which the driver requests notifications.

To register to receive notifications of session events, a kernel-mode driver calls the IoRegisterContainerNotification routine. To specify the session events for which the driver wants to receive notifications, the driver sets this structure's **EventMask** member to the bitwise OR of one or more **IO_SESSION_STATE_*XXX*** constants.

Set this member to the bitwise OR of one or more of the following **IO_SESSION_STATE_*XXX*** constants:

VALUE	MEANING

IO_SESSION_STATE_ALL_EVENTS 0xffffffff	Send notifications of all session events.
IO_SESSION_STATE_CREATION_EVENT 0x00000001	Send a notification when the user session is created.
IO_SESSION_STATE_TERMINATION_EVENT 0x00000002	Send a notification when the user session ends.

IO_SESSION_STATE_CONNECT_EVENT 0x00000004	Send a notification when the user session is connected.
IO_SESSION_STATE_DISCONNECT_EVENT 0x00000008	Send a notification when the user session is disconnected.
IO_SESSION_STATE_LOGON_EVENT 0x00000010	Send a notification when the user logs on to the session.

IO_SESSION_LOGOFF_EVENT 0x00000020	Send a notification when the user logs off of the session.
IO_SESSION_EVENT_ALL_EVENT_MASK 0x0000003f	Send a notification when any type of session event occurs.

`Context`

A pointer to a context buffer in which the driver can store its private data for a particular session notification registration. The I/O manager passes this pointer to the driver's notification callback routine (specified by the **IoRegisterContainerNotification** routine's *CallbackFunction* parameter). The I/O manager does not try to validate the *Context* pointer or to access the buffer that it points to. This member can be **NULL** if the driver does not require a context buffer.

Remarks

This structure is used by the **IoRegisterContainerNotification** routine. A driver that calls **IoRegisterContainerNotification** uses this structure to specify the session event notifications that it is registering for.

A per-session device object represents a device that can be accessed only by a particular user session. If a driver sets the **IoObject** member to point to a device object that is a per-session device object, **IoRegisterContainerNotification** registers the driver to receive only notifications of events that occur in the session that is represented by the device object. If **IoObject** points to a device object that is not a per-session device object, or if **IoObject** points to an object that is not a device object, **IoRegisterContainerNotification** registers the driver to receive notifications of events that occur in all sessions on the computer.

To determine whether a device object is a per-session device object, a driver can call the IoGetDevicePropertyData routine to query the DEVPKEY_Device_SessionId property key in the property store of the device object. If the **DEVPKEY_Device_SessionId** property exists and the value of the property is set to a nonzero Terminal Services session identifier, the device object is a per-session device object. For more information, see DEVPKEY_Device_SessionId.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

DEVICE_OBJECT

DEVPKEY_Device_SessionId

DRIVER_OBJECT

FILE_OBJECT

IoGetDevicePropertyData

IoRegisterContainerNotification

IO_STACK_LOCATION structure

2/5/2019 • 8 minutes to read • Edit Online

The **IO_STACK_LOCATION** structure defines an I/O stack location, which is an entry in the I/O stack that is associated with each IRP. Each I/O stack location in an IRP has some common members and some request-type-specific members.

Syntax

```
typedef struct _IO_STACK_LOCATION {
  UCHAR                 MajorFunction;
  UCHAR                 MinorFunction;
  UCHAR                 Flags;
  UCHAR                 Control;
  union {
    struct {
      PIO_SECURITY_CONTEXT     SecurityContext;
      ULONG                    Options;
      USHORT POINTER_ALIGNMENT FileAttributes;
      USHORT                   ShareAccess;
      ULONG POINTER_ALIGNMENT  EaLength;
    } Create;
    struct {
      PIO_SECURITY_CONTEXT          SecurityContext;
      ULONG                         Options;
      USHORT POINTER_ALIGNMENT      Reserved;
      USHORT                        ShareAccess;
      PNAMED_PIPE_CREATE_PARAMETERS Parameters;
    } CreatePipe;
    struct {
      PIO_SECURITY_CONTEXT        SecurityContext;
      ULONG                       Options;
      USHORT POINTER_ALIGNMENT    Reserved;
      USHORT                      ShareAccess;
      PMAILSLOT_CREATE_PARAMETERS Parameters;
    } CreateMailslot;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT Key;
      ULONG                   Flags;
      LARGE_INTEGER           ByteOffset;
    } Read;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT Key;
      ULONG                   Flags;
      LARGE_INTEGER           ByteOffset;
    } Write;
    struct {
      ULONG                   Length;
      PUNICODE_STRING         FileName;
      FILE_INFORMATION_CLASS  FileInformationClass;
      ULONG POINTER_ALIGNMENT FileIndex;
    } QueryDirectory;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT CompletionFilter;
    } NotifyDirectory;
    struct {
      ULONG                                                 Length;
      ULONG POINTER_ALIGNMENT                               CompletionFilter;
      DIRECTORY_NOTIFY_INFORMATION_CLASS POINTER_ALIGNMENT DirectoryNotifyInformationClass;
    } NotifyDirectoryEx;
    struct {
      ULONG                                    Length;
```

```
    FILE_INFORMATION_CLASS POINTER_ALIGNMENT FileInformationClass;
  } QueryFile;
  struct {
    ULONG                                   Length;
    FILE_INFORMATION_CLASS POINTER_ALIGNMENT FileInformationClass;
    PFILE_OBJECT                            FileObject;
    union {
      struct {
        BOOLEAN ReplaceIfExists;
        BOOLEAN AdvanceOnly;
      };
      ULONG  ClusterCount;
      HANDLE DeleteHandle;
    };
  } SetFile;
  struct {
    ULONG                   Length;
    PVOID                   EaList;
    ULONG                   EaListLength;
    ULONG POINTER_ALIGNMENT EaIndex;
  } QueryEa;
  struct {
    ULONG Length;
  } SetEa;
  struct {
    ULONG                                   Length;
    FS_INFORMATION_CLASS POINTER_ALIGNMENT FsInformationClass;
  } QueryVolume;
  struct {
    ULONG                                   Length;
    FS_INFORMATION_CLASS POINTER_ALIGNMENT FsInformationClass;
  } SetVolume;
  struct {
    ULONG                   OutputBufferLength;
    ULONG POINTER_ALIGNMENT InputBufferLength;
    ULONG POINTER_ALIGNMENT FsControlCode;
    PVOID                   Type3InputBuffer;
  } FileSystemControl;
  struct {
    PLARGE_INTEGER          Length;
    ULONG POINTER_ALIGNMENT Key;
    LARGE_INTEGER           ByteOffset;
  } LockControl;
  struct {
    ULONG                   OutputBufferLength;
    ULONG POINTER_ALIGNMENT InputBufferLength;
    ULONG POINTER_ALIGNMENT IoControlCode;
    PVOID                   Type3InputBuffer;
  } DeviceIoControl;
  struct {
    SECURITY_INFORMATION    SecurityInformation;
    ULONG POINTER_ALIGNMENT Length;
  } QuerySecurity;
  struct {
    SECURITY_INFORMATION SecurityInformation;
    PSECURITY_DESCRIPTOR SecurityDescriptor;
  } SetSecurity;
  struct {
    PVPB           Vpb;
    PDEVICE_OBJECT DeviceObject;
  } MountVolume;
  struct {
    PVPB           Vpb;
    PDEVICE_OBJECT DeviceObject;
  } VerifyVolume;
  struct {
    struct _SCSI_REQUEST_BLOCK *Srb;
  } Scsi;
  struct {
    ULONG                       Length;
    PSID                        StartSid;
    PFILE_GET_QUOTA_INFORMATION SidList;
```

```
      PFILE_GET_QUOTA_INFORMATION SidList;
      ULONG                       SidListLength;
    } QueryQuota;
    struct {
      ULONG Length;
    } SetQuota;
    struct {
      DEVICE_RELATION_TYPE Type;
    } QueryDeviceRelations;
    struct {
      const GUID *InterfaceType;
      USHORT     Size;
      USHORT     Version;
      PINTERFACE Interface;
      PVOID      InterfaceSpecificData;
    } QueryInterface;
    struct {
      PDEVICE_CAPABILITIES Capabilities;
    } DeviceCapabilities;
    struct {
      PIO_RESOURCE_REQUIREMENTS_LIST IoResourceRequirementList;
    } FilterResourceRequirements;
    struct {
      ULONG                  WhichSpace;
      PVOID                  Buffer;
      ULONG                  Offset;
      ULONG POINTER_ALIGNMENT Length;
    } ReadWriteConfig;
    struct {
      BOOLEAN Lock;
    } SetLock;
    struct {
      BUS_QUERY_ID_TYPE IdType;
    } QueryId;
    struct {
      DEVICE_TEXT_TYPE       DeviceTextType;
      LCID POINTER_ALIGNMENT LocaleId;
    } QueryDeviceText;
    struct {
      BOOLEAN                                          InPath;
      BOOLEAN                                          Reserved[3];
      DEVICE_USAGE_NOTIFICATION_TYPE POINTER_ALIGNMENT Type;
    } UsageNotification;
    struct {
      SYSTEM_POWER_STATE PowerState;
    } WaitWake;
    struct {
      PPOWER_SEQUENCE PowerSequence;
    } PowerSequence;
#if ...
    struct {
      union {
        ULONG                      SystemContext;
        SYSTEM_POWER_STATE_CONTEXT SystemPowerStateContext;
      };
      POWER_STATE_TYPE POINTER_ALIGNMENT Type;
      POWER_STATE POINTER_ALIGNMENT      State;
      POWER_ACTION POINTER_ALIGNMENT     ShutdownType;
    } Power;
#else
    struct {
      ULONG                              SystemContext;
      POWER_STATE_TYPE POINTER_ALIGNMENT Type;
      POWER_STATE POINTER_ALIGNMENT      State;
      POWER_ACTION POINTER_ALIGNMENT     ShutdownType;
    } Power;
#endif
    struct {
      PCM_RESOURCE_LIST AllocatedResources;
      PCM_RESOURCE_LIST AllocatedResourcesTranslated;
    } StartDevice;
    struct {
```

```
      ULONG_PTR ProviderId;
      PVOID     DataPath;
      ULONG     BufferSize;
      PVOID     Buffer;
    } WMI;
    struct {
      PVOID Argument1;
      PVOID Argument2;
      PVOID Argument3;
      PVOID Argument4;
    } Others;
  } Parameters;
  PDEVICE_OBJECT         DeviceObject;
  PFILE_OBJECT           FileObject;
  PIO_COMPLETION_ROUTINE CompletionRoutine;
  PVOID                  Context;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Members

`MajorFunction`

The IRP major function code indicating the type of I/O operation to be performed.

`MinorFunction`

A subfunction code for **MajorFunction**. The PnP manager, the power manager, file system drivers, and SCSI class drivers set this member for some requests.

`Flags`

Request-type-specific values used almost exclusively by file system drivers. Removable-media device drivers check whether this member is set with SL_OVERRIDE_VERIFY_VOLUME for read requests to determine whether to continue the read operation even if the device object's **Flags** is set with DO_VERIFY_VOLUME. Intermediate drivers layered over a removable-media device driver must copy this member into the I/O stack location of the next-lower driver in all incoming IRP_MJ_READ requests.

Possible flag values include:

FLAG	VALUE	DESCRIPTION
SL_KEY_SPECIFIED	`0x01`	Indicates that the `IO_STACK_LOCATION.Parameters.Read(OrWrite).K` contains which copy of a given sector should be read when redundancy is enabled. Today this flag is use only with IRP_MJ_READ operations.
SL_OVERRIDE_VERIFY_VOLUME	`0x02`	This flag is used to determine whether to continue the read operation even if the device object's **Flags** is set with DO_VERIFY_VOLUME.
SL_WRITE_THROUGH	`0x04`	This flag informs the storage driver to set appropriate flags so that the disk bypasses the write cache in order to force the disk to write through to its persistent storage media. **This flag is device-specific; not all disk drives support bypassing disk-cache.**
SL_FT_SEQUENTIAL_WRITE	`0x08`	Reserved for system use.

SL_FORCE_DIRECT_WRITE	`0x10`	This flag lets kernel-mode drivers write to volume areas that they normally cannot write to because of blocking direct write in the file system and storage driver stack. **This flag was introduced in Windows Vista.** **Note** Direct write blocking, introduced in Windows Vista helps improve security. This flag is checked both at the file system layer and storage stack layer. For more information about direct write blocking, see Blocking Direct Write Operations to Volumes and Disks.
SL_REALTIME_STREAM	`0x20`	This flag hints that the IO is for real-time streaming requests to a CD-ROM class driver. This hints the driver to perform READ/WRITE operations at a guaranteed speed for real-time streaming. **This flag is valid only with optical media.**
SL_PERSISTENT_MEMORY_FIXED_MAPPING	`0x20`	The persistent memory mapping of the bytes in the write request cannot change while handling this write request. **This flag is valid only with a persistent memory device and IRP_MJ_WRITE.**

Note (For persistent memory devices) One of the reasons for remapping (modifying the physical address of a given LBA) on persistent memory devices is to provide efficient sector level atomicity. If the flag is not set, remapping is allowed especially if it results in the driver providing sector atomicity. File systems (or the requester) prefer that a persistent memory device driver provides sector atomicity. If the flag is set, a persistent memory driver shall not remap the physical addresses corresponding to the LBAs. If that means sector atomicity can't be provided, so be it. However, the driver is more than welcome to provide sector atomicity as long as there is no remapping.

`Control`

Drivers can check this member to determine whether it is set with SL_PENDING_RETURNED. Drivers have read-only access to this member.

`Parameters`

A union that depends on the major and minor IRP function code values contained in **MajorFunction** and **MinorFunction**. The following table shows which IRPs use the individual members of the **Parameters** union.

MEMBER NAME	IRPS THAT USE THIS MEMBER
Create	IRP_MJ_CREATE
Read	IRP_MJ_READ
Write	IRP_MJ_WRITE
QueryFile	IRP_MJ_QUERY_INFORMATION
SetFile	IRP_MJ_SET_INFORMATION

QueryVolume	IRP_MJ_QUERY_VOLUME_INFORMATION
DeviceIoControl	IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL
MountVolume	**IRP_MN_MOUNT_VOLUME**
VerifyVolume	**IRP_MN_VERIFY_VOLUME**
Scsi	IRP_MJ_INTERNAL_DEVICE_CONTROL (SCSI)
QueryDeviceRelations	IRP_MN_QUERY_DEVICE_RELATIONS
QueryInterface	IRP_MN_QUERY_INTERFACE
DeviceCapabilities	IRP_MN_QUERY_CAPABILITIES
FilterResourceRequirements	IRP_MN_FILTER_RESOURCE_REQUIREMENTS
ReadWriteConfig	IRP_MN_READ_CONFIG and IRP_MN_WRITE_CONFIG
SetLock	IRP_MN_SET_LOCK
QueryId	IRP_MN_QUERY_ID
QueryDeviceText	IRP_MN_QUERY_DEVICE_TEXT
UsageNotification	IRP_MN_DEVICE_USAGE_NOTIFICATION
WaitWake	IRP_MN_WAIT_WAKE
PowerSequence	IRP_MN_POWER_SEQUENCE
Power	IRP_MN_SET_POWER and IRP_MN_QUERY_POWER
StartDevice	IRP_MN_START_DEVICE
WMI	WMI minor IRPs
Others	Driver-specific IRPs

For more information, see IRP Major Function Codes.

`Parameters.Create`

`Parameters.Create.SecurityContext`

`Parameters.Create.Options`

`Parameters.Create.FileAttributes`

`Parameters.Create.ShareAccess`

`Parameters.Create.EaLength`

`Parameters.CreatePipe`

`Parameters.CreatePipe.SecurityContext`

`Parameters.CreatePipe.Options`

`Parameters.CreatePipe.Reserved`

`Parameters.CreatePipe.ShareAccess`

`Parameters.CreatePipe.Parameters`

`Parameters.CreateMailslot`

`Parameters.CreateMailslot.SecurityContext`

`Parameters.CreateMailslot.Options`

`Parameters.CreateMailslot.Reserved`

`Parameters.CreateMailslot.ShareAccess`

`Parameters.CreateMailslot.Parameters`

`Parameters.Read`

`Parameters.Read.Length`

`Parameters.Read.Key`

`Parameters.Read.Flags`

`Parameters.Read.ByteOffset`

`Parameters.Write`

`Parameters.Write.Length`

`Parameters.Write.Key`

`Parameters.Write.Flags`

`Parameters.Write.ByteOffset`

`Parameters.QueryDirectory`

`Parameters.QueryDirectory.Length`

`Parameters.QueryDirectory.FileName`

`Parameters.QueryDirectory.FileInformationClass`

`Parameters.QueryDirectory.FileIndex`

`Parameters.NotifyDirectory`

`Parameters.NotifyDirectory.Length`

`Parameters.NotifyDirectory.CompletionFilter`

`Parameters.NotifyDirectoryEx`

`Parameters.NotifyDirectoryEx.Length`

`Parameters.NotifyDirectoryEx.CompletionFilter`

`Parameters.NotifyDirectoryEx.DirectoryNotifyInformationClass`

`Parameters.QueryFile`

`Parameters.QueryFile.Length`

`Parameters.QueryFile.FileInformationClass`

`Parameters.SetFile`

`Parameters.SetFile.Length`

`Parameters.SetFile.FileInformationClass`

`Parameters.SetFile.FileObject`

`Parameters.SetFile`

`Parameters.SetFile`

`Parameters.SetFile.ReplaceIfExists`

`Parameters.SetFile.AdvanceOnly`

`Parameters.SetFile.ClusterCount`

`Parameters.SetFile.DeleteHandle`

`Parameters.QueryEa`

`Parameters.QueryEa.Length`

`Parameters.QueryEa.EaList`

`Parameters.QueryEa.EaListLength`

`Parameters.QueryEa.EaIndex`

`Parameters.SetEa`

`Parameters.SetEa.Length`

`Parameters.QueryVolume`

`Parameters.QueryVolume.Length`

`Parameters.QueryVolume.FsInformationClass`

`Parameters.SetVolume`

`Parameters.SetVolume.Length`

`Parameters.SetVolume.FsInformationClass`

`Parameters.FileSystemControl`

`Parameters.FileSystemControl.OutputBufferLength`

`Parameters.FileSystemControl.InputBufferLength`

`Parameters.FileSystemControl.FsControlCode`

`Parameters.FileSystemControl.Type3InputBuffer`

`Parameters.LockControl`

`Parameters.LockControl.Length`

`Parameters.LockControl.Key`

`Parameters.LockControl.ByteOffset`

`Parameters.DeviceIoControl`

`Parameters.DeviceIoControl.OutputBufferLength`

`Parameters.DeviceIoControl.InputBufferLength`

`Parameters.DeviceIoControl.IoControlCode`

`Parameters.DeviceIoControl.Type3InputBuffer`

`Parameters.QuerySecurity`

`Parameters.QuerySecurity.SecurityInformation`

`Parameters.QuerySecurity.Length`

`Parameters.SetSecurity`

`Parameters.SetSecurity.SecurityInformation`

`Parameters.SetSecurity.SecurityDescriptor`

`Parameters.MountVolume`

`Parameters.MountVolume.Vpb`

`Parameters.MountVolume.DeviceObject`

`Parameters.VerifyVolume`

`Parameters.VerifyVolume.Vpb`

`Parameters.VerifyVolume.DeviceObject`

`Parameters.Scsi`

`Parameters.Scsi.Srb`

`Parameters.QueryQuota`

`Parameters.QueryQuota.Length`

`Parameters.QueryQuota.StartSid`

`Parameters.QueryQuota.SidList`

`Parameters.QueryQuota.SidListLength`

`Parameters.SetQuota`

`Parameters.SetQuota.Length`

`Parameters.QueryDeviceRelations`

`Parameters.QueryDeviceRelations.Type`

`Parameters.QueryInterface`

`Parameters.QueryInterface.InterfaceType`

`Parameters.QueryInterface.Size`

`Parameters.QueryInterface.Version`

`Parameters.QueryInterface.Interface`

`Parameters.QueryInterface.InterfaceSpecificData`

`Parameters.DeviceCapabilities`

Parameters.DeviceCapabilities.Capabilities

Parameters.FilterResourceRequirements

Parameters.FilterResourceRequirements.IoResourceRequirementList

Parameters.ReadWriteConfig

Parameters.ReadWriteConfig.WhichSpace

Parameters.ReadWriteConfig.Buffer

Parameters.ReadWriteConfig.Offset

Parameters.ReadWriteConfig.Length

Parameters.SetLock

Parameters.SetLock.Lock

Parameters.QueryId

Parameters.QueryId.IdType

Parameters.QueryDeviceText

Parameters.QueryDeviceText.DeviceTextType

Parameters.QueryDeviceText.LocaleId

Parameters.UsageNotification

Parameters.UsageNotification.InPath

Parameters.UsageNotification.Reserved

Parameters.UsageNotification.Type

Parameters.WaitWake

Parameters.WaitWake.PowerState

Parameters.PowerSequence

Parameters.PowerSequence.PowerSequence

Parameters.Power.#2398894558

Parameters.Power.#2398894558

Parameters.Power.#2398894558.SystemContext

Parameters.Power.#2398894558.SystemPowerStateContext

Parameters.Power.#2398894558.Type

Parameters.Power.#2398894558.State

Parameters.Power.#2398894558.ShutdownType

Parameters.Power.#1950259185

Parameters.Power.#1950259185.SystemContext

Parameters.Power.#1950259185.Type

Parameters.Power.#1950259185.State

`Parameters.Power.#1950259185.ShutdownType`

`Parameters.StartDevice`

`Parameters.StartDevice.AllocatedResources`

`Parameters.StartDevice.AllocatedResourcesTranslated`

`Parameters.WMI`

`Parameters.WMI.ProviderId`

`Parameters.WMI.DataPath`

`Parameters.WMI.BufferSize`

`Parameters.WMI.Buffer`

`Parameters.Others`

`Parameters.Others.Argument1`

`Parameters.Others.Argument2`

`Parameters.Others.Argument3`

`Parameters.Others.Argument4`

`DeviceObject`

A pointer to the driver-created DEVICE_OBJECT structure representing the target physical, logical, or virtual device for which this driver is to handle the IRP.

`FileObject`

A pointer to a FILE_OBJECT structure that represents the file object, if any, that is associated with **DeviceObject** pointer.

`CompletionRoutine`

The completion routines that is invoked depending on the flags in the preceding flags field.

`Context`

Driver-defined context that is used to store the address of the context parameter that is passed to the CompletionRoutine.

Remarks

For each IRP, there is one **IO_STACK_LOCATION** structure for each driver in a driver stack. Each IRP's set of I/O stack locations is appended to the IRP, following the IRP structure.

Every higher-level driver is responsible for setting up the I/O stack location for the next-lower driver in each IRP. A driver must call IoGetCurrentIrpStackLocation to get a pointer to its own stack location for each IRP. Higher-level drivers can call IoGetNextIrpStackLocation to get a pointer to the next-lower driver's stack location.

The higher-level driver must set up the stack location contents before calling IoCallDriver to pass an IRP to the lower-level driver. If the driver will pass the input IRP on to the next lower-level driver, the dispatch routine should call IoSkipCurrentIrpStackLocation or IoCopyCurrentIrpStackLocationToNext to set up the I/O stack location of the next-lower driver.

A higher-level driver's call to **IoCallDriver** sets the **DeviceObject** member to the next-lower-level driver's target device object, in the I/O stack location of the lower driver. The I/O manager passes each higher-level driver's IoCompletion routine a pointer to its own device object when the *IoCompletion* routine is called on completion of the

IRP.

If a higher-level driver allocates IRPs to make requests of its own, its *IoCompletion* routine is passed a **NULL** **DeviceObject** pointer if that driver neither allocates a stack location for itself nor sets up the **DeviceObject** pointer in its own stack location of the newly allocated IRP.

In some cases, a higher-level driver layered over a mass-storage device driver is responsible for splitting up large transfer requests for the underlying device driver. In particular, SCSI class drivers must check the **Parameters.Read.Length** and **Parameters.Write.Length**, determine whether the size of the requested transfer exceeds the underlying HBA's transfer capabilities, and, if so, split the **Length** of the original request into a sequence of partial transfers to satisfy the original IRP.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_STATUS_BLOCK

IRP

IoCallDriver

IoCopyCurrentIrpStackLocationToNext

IoGetCurrentIrpStackLocation

IoGetNextIrpStackLocation

IoSetCompletionRoutine

IoSetNextIrpStackLocation

IoSkipCurrentIrpStackLocation

IO_STATUS_BLOCK structure

1/30/2019 • 2 minutes to read • Edit Online

A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.

Syntax

```
typedef struct _IO_STATUS_BLOCK {
  union {
    NTSTATUS Status;
    PVOID    Pointer;
  } DUMMYUNIONNAME;
  ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Status`

This is the completion status, either STATUS_SUCCESS if the requested operation was completed successfully or an informational, warning, or error STATUS_*XXX* value. For more information, see Using NTSTATUS values.

`DUMMYUNIONNAME.Pointer`

Reserved. For internal use only.

`Information`

This is set to a request-dependent value. For example, on successful completion of a transfer request, this is set to the number of bytes transferred. If a transfer request is completed with another STATUS_*XXX*, this member is set to zero.

Remarks

Unless a driver's dispatch routine completes an IRP with an error status value, the lowest-level driver in the chain frequently sets the IRP's I/O status block to the values that will be returned to the original requester of the I/O operation.

The IoCompletion routines of higher-level drivers usually check the I/O status block in IRPs completed by lower drivers. By design, the I/O status block in an IRP is the only information passed back from the underlying device driver to all higher-level drivers' *IoCompletion* routines.

The operating system implements support routines that write **IO_STATUS_BLOCK** values to caller-supplied output buffers. For example, see ZwOpenFile or NtOpenFile. These routines return status codes that might not match the status codes in the **IO_STATUS_BLOCK** structures. If one of these routines returns STATUS_PENDING, the caller should wait for the I/O operation to complete, and then check the status code in the **IO_STATUS_BLOCK** structure to determine the final status of the operation. If the routine returns a status code other than STATUS_PENDING, the caller should rely on this status code instead of the status code in the **IO_STATUS_BLOCK** structure.

For more information, see I/O Status Blocks.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

IO_STACK_LOCATION

IRP

IoCompleteRequest

IoSetCompletionRoutine

IRP structure

2/5/2019 • 6 minutes to read • Edit Online

The **IRP** structure is a partially opaque structure that represents an *I/O request packet*. Drivers can use the following members of the IRP structure.

Syntax

```
typedef struct _IRP {
  CSHORT                    Type;
  USHORT                    Size;
  PMDL                      MdlAddress;
  ULONG                     Flags;
  union {
    struct _IRP     *MasterIrp;
    __volatile LONG IrpCount;
    PVOID           SystemBuffer;
  } AssociatedIrp;
  LIST_ENTRY                ThreadListEntry;
  IO_STATUS_BLOCK           IoStatus;
  KPROCESSOR_MODE           RequestorMode;
  BOOLEAN                   PendingReturned;
  CHAR                      StackCount;
  CHAR                      CurrentLocation;
  BOOLEAN                   Cancel;
  KIRQL                     CancelIrql;
  CCHAR                     ApcEnvironment;
  UCHAR                     AllocationFlags;
  PIO_STATUS_BLOCK          UserIosb;
  PKEVENT                   UserEvent;
  union {
    struct {
      union {
        PIO_APC_ROUTINE UserApcRoutine;
        PVOID           IssuingProcess;
      };
      PVOID UserApcContext;
    } AsynchronousParameters;
    LARGE_INTEGER AllocationSize;
  } Overlay;
  __volatile PDRIVER_CANCEL CancelRoutine;
  PVOID                     UserBuffer;
  union {
    struct {
      union {
        KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
        struct {
          PVOID DriverContext[4];
        };
      };
      PETHREAD     Thread;
      PCHAR        AuxiliaryBuffer;
      struct {
        LIST_ENTRY ListEntry;
        union {
          struct _IO_STACK_LOCATION *CurrentStackLocation;
          ULONG                     PacketType;
        };
      };
      PFILE_OBJECT OriginalFileObject;
    } Overlay;
    KAPC  Apc;
    PVOID CompletionKey;
  } Tail;
} IRP;
```

Members

`Type`

`Size`

`MdlAddress`

Pointer to an MDL describing a user buffer, if the driver is using direct I/O, and the IRP major function code is one of the following:

IRP_MJ_READ

The MDL describes an empty buffer that the device or driver fills in.

IRP_MJ_WRITE

The MDL describes a buffer that contains data for the device or driver.

IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL

If the IOCTL code specifies the METHOD_IN_DIRECT transfer type, the MDL describes a buffer that contains data for the device or driver.

If the IOCTL code specifies the METHOD_OUT_DIRECT transfer type, the MDL describes an empty buffer that the device or driver fills in.

For more information about the buffers that are associated with METHOD_IN_DIRECT and METHOD_OUT_DIRECT transfer types in IOCTL codes, see Buffer Descriptions for I/O Control Codes.

If the driver is not using direct I/O, this pointer is **NULL**.

`Flags`

File system drivers use this field, which is read-only for all drivers. Network and, possibly, highest-level device drivers also might read this field. This field is set either to zero or to the bitwise-OR of one or more of the following system-defined flag bits:

IRP_NOCACHE

IRP_PAGING_IO

IRP_MOUNT_COMPLETION

IRP_SYNCHRONOUS_API

IRP_ASSOCIATED_IRP

IRP_BUFFERED_IO

IRP_DEALLOCATE_BUFFER

IRP_INPUT_OPERATION

IRP_SYNCHRONOUS_PAGING_IO

IRP_CREATE_OPERATION

IRP_READ_OPERATION

IRP_WRITE_OPERATION

IRP_CLOSE_OPERATION

IRP_DEFER_IO_COMPLETION

IRP_OB_QUERY_NAME

IRP_HOLD_DEVICE_QUEUE

IRP_UM_DRIVER_INITIATED_IO

`AssociatedIrp`

`AssociatedIrp.MasterIrp`

Pointer to the master IRP in an IRP that was created by a highest-level driver's call to IoMakeAssociatedIrp.

`AssociatedIrp.IrpCount`

`AssociatedIrp.SystemBuffer`

Pointer to a system-space buffer.

If the driver is using buffered I/O, the buffer's purpose is determined by the IRP major function code, as follows:

SystemBuffer.IRP_MJ_READ

The buffer receives data from the device or driver. The buffer's length is specified by **Parameters.Read.Length** in the driver's IO_STACK_LOCATION structure.

NULL.

SystemBuffer.IRP_MJ_WRITE

The buffer supplies data for the device or driver. The buffer's length is specified by **Parameters.Write.Length** in the driver's **IO_STACK_LOCATION** structure.

NULL.

SystemBuffer.IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL

The buffer represents both the input and output buffers that are supplied to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**. Output data overwrites input data.

For input, the buffer's length is specified by **Parameters.DeviceIoControl.InputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

For output, the buffer's length is specified by **Parameters.DeviceIoControl.OutputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

For more information, see Buffer Descriptions for I/O Control Codes.

The buffer represents the input buffer that is supplied to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**.

The buffer's length is specified by **Parameters.DeviceIoControl.InputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

For more information, see Buffer Descriptions for I/O Control Codes.

If the driver is using direct I/O, the buffer's purpose is determined by the IRP major function code, as follows:

`ThreadListEntry`

`IoStatus`

Contains the IO_STATUS_BLOCK structure in which a driver stores status and information before calling IoCompleteRequest.

`RequestorMode`

Indicates the execution mode of the original requester of the operation, one of **UserMode** or **KernelMode**.

`PendingReturned`

If set to **TRUE**, a driver has marked the IRP pending. Each IoCompletion routine should check the value of this flag. If the flag is **TRUE**, and if the IoCompletion routine will not return STATUS_MORE_PROCESSING_REQUIRED, the routine should call IoMarkIrpPending to propagate the pending status to drivers above it in the device stack.

`StackCount`

`CurrentLocation`

`Cancel`

If set to **TRUE**, the IRP either is or should be canceled.

`CancelIrql`

Contains the IRQL at which a driver is running when IoAcquireCancelSpinLock is called.

`ApcEnvironment`

`AllocationFlags`

`UserIosb`

`UserEvent`

`Overlay`

`Overlay.AsynchronousParameters`

`Overlay.AsynchronousParameters`

`Overlay.AsynchronousParameters.UserApcRoutine`

`Overlay.AsynchronousParameters.IssuingProcess`

`Overlay.AsynchronousParameters.UserApcContext`

`Overlay.AllocationSize`

`CancelRoutine`

Contains the entry point for a driver-supplied Cancel routine to be called if the IRP is canceled. **NULL** indicates that the IRP is not currently cancelable.

`UserBuffer`

Contains the address of an output buffer if both of the following conditions apply:

- The major function code in the I/O stack location is IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL.
- The I/O control code was defined with METHOD_NEITHER or METHOD_BUFFERED.

For METHOD_BUFFERED, the driver should use the buffer pointed to by **Irp->AssociatedIrp.SystemBuffer** as the output buffer. When the driver completes the request, the I/O manager copies the contents of this buffer to the output buffer that is pointed to by **Irp->UserBuffer**. The driver should not write directly to the buffer pointed to by **Irp->UserBuffer**. For more information, see Buffer Descriptions for I/O Control Codes.

`Tail`

`Tail.Overlay`

`Tail.Overlay`

`Tail.Overlay.DeviceQueueEntry`

If IRPs are queued in the device queue associated with the driver's device object, this field links IRPs in the device queue. These links can be used only while the driver is processing the IRP.

`Tail.Overlay`

`Tail.Overlay.DriverContext`

If IRPs are not queued in the device queue associated with the driver's device object, this field can be used by the driver to store up to four pointers. This field can be used only while the driver owns the IRP.

`Tail.Overlay.Thread`

A pointer to the caller's thread control block (TCB). For requests that originate in user-mode, the I/O manager always sets this field to point to the TCB of the thread that issued the request.

`Tail.Overlay.AuxiliaryBuffer`

`Tail.Overlay`

`Tail.Overlay.ListEntry`

If a driver manages its own internal queues of IRPs, it uses this field to link one IRP to the next. These links can be used only while the driver is holding the IRP in its queue or is processing the IRP.

`Tail.Overlay`

`Tail.Overlay.CurrentStackLocation`

`Tail.Overlay.PacketType`

`Tail.Overlay.OriginalFileObject`

`Tail.Apc`

`Tail.CompletionKey`

Remarks

Undocumented members of the IRP structure are reserved, used only by the I/O manager or, in some cases, by FSDs.

An IRP is the basic I/O manager structure used to communicate with drivers and to allow drivers to communicate with each other. A packet consists of two different parts:

- *Header*, or *fixed part of the packet*— This is used by the I/O manager to store information about the original request, such as the caller's device-independent parameters, the address of the device object upon which a file is open, and so on. It is also used by drivers to store information such as the final status of the request.
- *I/O stack locations*— Following the header is a set of I/O stack locations, one per driver in the chain of layered drivers for which the request is bound. Each stack location contains the parameters, function codes, and context used by the corresponding driver to determine what it is supposed to be doing. For more information, see the IO_STACK_LOCATION structure.

While a higher-level driver might check the value of the **Cancel** Boolean in an IRP, that driver cannot assume the IRP will be completed with STATUS_CANCELLED by a lower-level driver even if the value is **TRUE**.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IO_STACK_LOCATION

IO_STATUS_BLOCK

IoCreateDevice

IoGetCurrentIrpStackLocation

IoGetNextIrpStackLocation

IoSetCancelRoutine

IoSetNextIrpStackLocation

IRQ_DEVICE_POLICY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IRQ_DEVICE_POLICY** enumeration type indicates the policy the operating system can use to assign the interrupts from a device to different processors.

Syntax

```
typedef enum _IRQ_DEVICE_POLICY {
  IrqPolicyMachineDefault,
  IrqPolicyAllCloseProcessors,
  IrqPolicyOneCloseProcessor,
  IrqPolicyAllProcessorsInMachine,
  IrqPolicySpecifiedProcessors,
  IrqPolicySpreadMessagesAcrossAllProcessors,
  IrqPolicyAllProcessorsInMachineWhenSteered
} IRQ_DEVICE_POLICY, *PIRQ_DEVICE_POLICY;
```

Constants

IrqPolicyMachineDefault	The device does not require any particular assignment of interrupts to processors.
IrqPolicyAllCloseProcessors	The operating system should assign interrupts from the device to processors that are close to the device. On non-NUMA computers, the effect of this value is identical to that of **IrqPolicyAllProcessorsInMachine**.
IrqPolicyOneCloseProcessor	The operating system should assign a single interrupt for the device to one processor that is close to the device. On non-NUMA computers, the operating system can assign the interrupt to any processor.
IrqPolicyAllProcessorsInMachine	The operating system should assign interrupts from the device to all processors.
IrqPolicySpecifiedProcessors	The operating system should assign interrupts from the device to a specific set of processors.
IrqPolicySpreadMessagesAcrossAllProcessors	The operating system should assign different message-signaled interrupts to different processors, if possible.
IrqPolicyAllProcessorsInMachineWhenSteered	

Remarks

The **Interrupt.AffinityPolicy** member of the IO_RESOURCE_DESCRIPTOR structure is an **IRQ_DEVICE_POLICY** enumeration value.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

IRQ_PRIORITY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **IRQ_PRIORITY** enumeration type indicates the priority the system should give to servicing a device's interrupts.

Syntax

```
typedef enum _IRQ_PRIORITY {
  IrqPriorityUndefined,
  IrqPriorityLow,
  IrqPriorityNormal,
  IrqPriorityHigh
} IRQ_PRIORITY, *PIRQ_PRIORITY;
```

Constants

IrqPriorityUndefined	Specifies that the device does not require any particular priority for its interrupts.
IrqPriorityLow	Specifies that the device's interrupts are of low priority. This setting is appropiate for devices that can tolerate higher-than-normal latency.
IrqPriorityNormal	Specifies that the device's interrupts are of normal priority.
IrqPriorityHigh	Specifies that the device's interrupts are of high priority. This setting is appropriate for devices that require low latency.

Remarks

The system uses **IRQ_PRIORITY** to assign IRQLs for devices. For example, it might assign a higher IRQL to a device that has an **IRQ_PRIORITY** of **IrqPriorityHigh** than it does to a device that has an **IRQ_PRIORITY** of **IrqPriorityLow**.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

IO_RESOURCE_DESCRIPTOR

KBUGCHECK_ADD_PAGES structure

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_ADD_PAGES** structure describes one or more pages of driver-supplied data to be written by a BugCheckAddPagesCallback callback routine to the crash dump file.

Syntax

```
typedef struct _KBUGCHECK_ADD_PAGES {
  PVOID     Context;
  ULONG     Flags;
  ULONG     BugCheckCode;
  ULONG_PTR Address;
  ULONG_PTR Count;
} KBUGCHECK_ADD_PAGES, *PKBUGCHECK_ADD_PAGES;
```

Members

`Context`

Contains private context data for the exclusive use of the callback routine. The callback routine can set this member to any value. Typically, if the callback routine needs to be called more than one time, the routine sets this member to point to a driver-supplied buffer during the initial call. During subsequent calls, the callback routine can read the previous contents of this buffer and update its contents. Before the initial call to the callback routine, **Context** is **NULL**.

`Flags`

Contains flags that describe the add-page request. The callback routine must set the value of this member. Set this member to the bitwise OR of one or more of the following flag bits:

KB_ADD_PAGES_FLAG_VIRTUAL_ADDRESS

Indicates that the **Address** member contains a virtual address.

KB_ADD_PAGES_FLAG_PHYSICAL_ADDRESS

Indicates that the **Address** member contains a physical address.

KB_ADD_PAGES_FLAG_ADDITIONAL_RANGES_EXIST

Indicates that the callback routine requests that it be called again so that it can add more pages.

The callback routine must set either the KB_ADD_PAGES_FLAG_VIRTUAL_ADDRESS flag or the KB_ADD_PAGES_FLAG_PHYSICAL_ADDRESS flag, but not both. On entry to the callback routine, **Flags** is initialized to zero.

`BugCheckCode`

Contains a bug check code, which specifies the reason for the bug check. The callback routine can use this information to decide whether to add any pages to the crash dump file. For a full list of bug check codes, see the Bugcodes.h header file.

`Address`

Specifies the physical or virtual address of the page or pages that the callback routine requests be added to the crash dump file.

`Count`

Specifies the number of contiguous pages to add to the crash dump file, starting from the virtual or physical address that is specified by the **Address** member. If **Count** > 1 and **Address** is a virtual address, the pages are contiguous in virtual memory space. If **Count** > 1 and **Address** is a physical address, the pages are contiguous in physical memory space. The callback routine can set this member to zero to indicate that it does not need to add any pages to the crash dump file.

Remarks

In a call to the *BugCheckAddPagesCallback* callback routine, the operating system sets the *Reason* parameter to **KbCallbackAddPages**, and sets the *ReasonSpecificData* parameter to point to a **KBUGCHECK_ADD_PAGES** structure.

For more information about how this structure is used, see BugCheckAddPagesCallback.

Requirements

Minimum supported client	Supported in Windows Server 2008 and later versions of Windows.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BugCheckAddPagesCallback

KBUGCHECK_CALLBACK_REASON Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_CALLBACK_REASON** enumeration type specifies the situations in which a bug-check callback executes.

Syntax

```
typedef enum _KBUGCHECK_CALLBACK_REASON {
  KbCallbackInvalid,
  KbCallbackReserved1,
  KbCallbackSecondaryDumpData,
  KbCallbackDumpIo,
  KbCallbackAddPages,
  KbCallbackSecondaryMultiPartDumpData,
  KbCallbackRemovePages,
  KbCallbackTriageDumpData
} KBUGCHECK_CALLBACK_REASON;
```

Constants

KbCallbackInvalid	Reserved for system use. Do not use.
KbCallbackReserved1	Reserved for system use. Do not use.
KbCallbackSecondaryDumpData	Specifies that the callback is executed to provide data that the system appends to the secondary section of the crash dump file. For more information about this type of callback, see BugCheckSecondaryDumpDataCallback.
KbCallbackDumpIo	Specifies that the callback is executed each time a section of the dump file is written. For more information about this type of callback, see BugCheckDumpIoCallback.
KbCallbackAddPages	Specifies that the callback is executed to provide one or more pages of data that the system adds to the primary section of the crash dump file. For more information about this type of callback, see BugCheckAddPagesCallback. This enumeration value is supported in Windows Server 2008 and later versions of Windows.
KbCallbackSecondaryMultiPartDumpData	Specifies that the callback is executed to get the amount of data the driver wants to store in the dump file. For more information about this type of callback, see BugCheckAddPagesCallback. This enumeration value is supported in Windows Server 2008 and later versions of Windows.
KbCallbackRemovePages	
KbCallbackTriageDumpData	

Requirements

Minimum supported client	Supported on Windows XP with Service Pack 1 (SP1), Windows Server 2003, and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BugCheckAddPagesCallback

BugCheckDumpIoCallback

BugCheckSecondaryDumpDataCallback

KBUGCHECK_REASON_CALLBACK_RECORD

KeRegisterBugCheckReasonCallback

KBUGCHECK_DUMP_IO structure

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_DUMP_IO** structure describes an I/O operation on the crash dump file.

Syntax

```
typedef struct _KBUGCHECK_DUMP_IO {
  IN ULONG64                Offset;
  IN PVOID                  Buffer;
  IN ULONG                  BufferLength;
  IN KBUGCHECK_DUMP_IO_TYPE Type;
} KBUGCHECK_DUMP_IO, *PKBUGCHECK_DUMP_IO;
```

Members

`Offset`

Specifies the current offset in the crash dump file, or -1 if the crash dump file is being written sequentially.

`Buffer`

Pointer to a buffer that contains the current data to be written to the dump file.

`BufferLength`

Specifies the length of the buffer, in bytes, that is specified by the **Buffer** member.

`Type`

Specifies the KBUGCHECK_DUMP_IO_TYPE value that signifies the type of data to be written to the dump file.

Remarks

For information about how this structure is used, see BugCheckDumpIoCallback.

Requirements

Minimum supported client	Available on Microsoft Windows XP with Service Pack 1 (SP1), Windows Server 2003, and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BugCheckDumpIoCallback

KBUGCHECK_DUMP_IO_TYPE

KBUGCHECK_DUMP_IO_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_DUMP_IO_TYPE** enumeration type identifies the type of a section of data within a crash dump file.

Syntax

```
typedef enum _KBUGCHECK_DUMP_IO_TYPE {
  KbDumpIoInvalid,
  KbDumpIoHeader,
  KbDumpIoBody,
  KbDumpIoSecondaryData,
  KbDumpIoComplete
} KBUGCHECK_DUMP_IO_TYPE;
```

Constants

KbDumpIoInvalid	Reserved for system use. Do not use.
KbDumpIoHeader	Specifies that crash dump data is header information.
KbDumpIoBody	Specifies that the crash dump data is part of the main body of the crash dump, such as the memory state at the time of the bug check.
KbDumpIoSecondaryData	Specifies that the crash dump data is data returned by a BugCheckSecondaryDumpDataCallback routine.
KbDumpIoComplete	Specifies that the crash dump data has been completely written.

Remarks

KBUGCHECK_DUMP_IO_TYPE values are used in the **Type** member of KBUGCHECK_DUMP_IO to specify the type of data in an I/O operation on the crash dump file.

For more information about how this enumeration type is used, see BugCheckDumpIoCallback.

Requirements

Minimum supported client	Available on Microsoft Windows XP with Service Pack 1 (SP1), Windows Server 2003, and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BugCheckDumpIoCallback

BugCheckSecondaryDumpDataCallback

KBUGCHECK_DUMP_IO

KBUGCHECK_SECONDARY_DUMP_DATA structure

1/30/2019 • 2 minutes to read • Edit Online

The **KBUGCHECK_SECONDARY_DUMP_DATA** structure describes a section of driver-supplied data to be written by BugCheckSecondaryDumpDataCallback to the crash dump file.

Syntax

```
typedef struct _KBUGCHECK_SECONDARY_DUMP_DATA {
  IN PVOID  InBuffer;
  IN ULONG  InBufferLength;
  IN ULONG  MaximumAllowed;
  OUT GUID  Guid;
  OUT PVOID OutBuffer;
  OUT ULONG OutBufferLength;
} KBUGCHECK_SECONDARY_DUMP_DATA, *PKBUGCHECK_SECONDARY_DUMP_DATA;
```

Members

`InBuffer`

Pointer to a buffer that is allocated by the system.

`InBufferLength`

Specifies the size of the buffer, in bytes, specified by the **InBuffer** member.

`MaximumAllowed`

Specifies the maximum amount of data that the *BugCheckSecondaryDumpDataCallback* routine can write to the crash dump file.

`Guid`

Specifies a GUID that identifies the driver's crash dump data. (Drivers must use unique GUIDs to mark their crash dump data. Use the GuidGen.exe tool to generate GUIDs for your driver. This tool is included in the Microsoft Windows SDK.)

`OutBuffer`

Pointer to the buffer where the driver writes its crash dump data, or **NULL**.

`OutBufferLength`

Specifies the size of the buffer, in bytes, that was specified by the **OutBuffer** member.

Remarks

For more information about how this structure is used, see BugCheckSecondaryDumpDataCallback.

Requirements

Minimum supported client	Available on Microsoft Windows XP with Service Pack 1 (SP1), Windows Server 2003, and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

BugCheckSecondaryDumpDataCallback

KDPC_WATCHDOG_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KDPC_WATCHDOG_INFORMATION** structure holds time-out information about the current deferred procedure call (DPC).

Syntax

```
typedef struct _KDPC_WATCHDOG_INFORMATION {
  ULONG DpcTimeLimit;
  ULONG DpcTimeCount;
  ULONG DpcWatchdogLimit;
  ULONG DpcWatchdogCount;
  ULONG Reserved;
} KDPC_WATCHDOG_INFORMATION, *PKDPC_WATCHDOG_INFORMATION;
```

Members

`DpcTimeLimit`

Time limit for a single, current deferred procedure call. If DPC time-out has been disabled, this value is set to 0.

`DpcTimeCount`

Time remaining for the current deferred procedure call, if DPC time-out has been enabled.

`DpcWatchdogLimit`

Total time limit permitted for a sequence of deferred procedure calls. If DPC watchdog has been disabled, this value is set to zero.

`DpcWatchdogCount`

Time value remaining for the current sequence of deferred procedure calls, if enabled.

`Reserved`

Reserved for system use.

Remarks

Time figures stored in each member are dimensionless.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KeQueryDpcWatchdogInformation

KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT** structure describes the notification context that is passed to a registered callback function when a new processor is dynamically added to a hardware partition.

Syntax

```
typedef struct _KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT {
  KE_PROCESSOR_CHANGE_NOTIFY_STATE State;
  ULONG                            NtNumber;
  NTSTATUS                         Status;
  PROCESSOR_NUMBER                 ProcNumber;
} KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT, *PKE_PROCESSOR_CHANGE_NOTIFY_CONTEXT;
```

Members

`State`

The state of the processor add operation. Possible values are as follows:

KeProcessorAddStartNotify

The operating system is about to add the processor. At this state, a device driver that receives this notification can allocate any per-processor data structures and perform any other required tasks to prepare the driver for execution on the new processor.

KeProcessorAddCompleteNotify

The operating system has successfully added the processor. At this state, a device driver that receives this notification can start scheduling threads on the new processor.

KeProcessorAddFailureNotify

The operating system failed to add the processor. If a device driver receives this notification, it should free any per-processor data structures that it allocated for the new processor when it received the **KeProcessorAddStartNotify** notification.

`NtNumber`

The processor index of the new processor. For information about the relationship between a processor index and a processor number, see KeGetProcessorNumberFromIndex.

`Status`

If the **State** member contains **KeProcessorAddFailureNotify**, this member contains the error status that indicates why the processor add operation failed.

`ProcNumber`

The processor number of the new processor. This member is a PROCESSOR_NUMBER structure that specifies a group number and a group-relative processor number.

Remarks

The **KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT** structure is available starting with Windows Server 2008.

A device driver registers to receive notification when a new processor is dynamically added to the hardware partition by calling the KeRegisterProcessorChangeCallback function. For more information about registering for this notification, see Registering for Synchronous Driver Notification.

When a new processor is dynamically added to the hardware partition, the operating system calls each registered callback function two times. The operating system calls each registered callback function the first time with the **KeProcessorAddStartNotify** state, and the second time with either the **KeProcessorAddCompleteNotify** state or the **KeProcessorAddFailureNotify** state.

A device driver's callback function that receives these notifications must not modify the contents of the **KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT** structure.

Requirements

Minimum supported client	Supported starting with Windows Server 2008.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KeGetProcessorNumberFromIndex

KeRegisterProcessorChangeCallback

PROCESSOR_NUMBER

KEY_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_BASIC_INFORMATION** structure defines a subset of the full information that is available for a registry key.

Syntax

```
typedef struct _KEY_BASIC_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG         TitleIndex;
  ULONG         NameLength;
  WCHAR         Name[1];
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;
```

Members

`LastWriteTime`

The last time this key or any of its values changed. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

`TitleIndex`

Device and intermediate drivers should ignore this member.

`NameLength`

The size, in bytes, of the key name string in the **Name** array.

`Name`

An array of wide characters that contains the name of the registry key. This character string is not null-terminated. Only the first element in this array is included in the **KEY_BASIC_INFORMATION** structure definition. The storage for the remaining elements in the array immediately follows this element.

Remarks

The ZwEnumerateKey and ZwQueryKey routines use the **KEY_BASIC_INFORMATION** structure to contain the basic information for a registry key. When the *KeyInformationClass* parameter of either routine is **KeyBasicInformation**, the *KeyInformation* buffer is treated as a **KEY_BASIC_INFORMATION** structure. For more information about the **KeyBasicInformation** enumeration value, see KEY_INFORMATION_CLASS.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_CACHED_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwEnumerateKey

ZwQueryKey

KEY_FULL_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_FULL_INFORMATION** structure defines the information available for a registry key, including information about its subkeys and the maximum length for their names and value entries. This information can be used to size buffers to get the names of subkeys and their value entries.

Syntax

```
typedef struct _KEY_FULL_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG         TitleIndex;
  ULONG         ClassOffset;
  ULONG         ClassLength;
  ULONG         SubKeys;
  ULONG         MaxNameLen;
  ULONG         MaxClassLen;
  ULONG         Values;
  ULONG         MaxValueNameLen;
  ULONG         MaxValueDataLen;
  WCHAR         Class[1];
} KEY_FULL_INFORMATION, *PKEY_FULL_INFORMATION;
```

Members

`LastWriteTime`

The last time this key or any of its values changed. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

`TitleIndex`

Device and intermediate drivers should ignore this member.

`ClassOffset`

The byte offset from the start of this structure to the **Class** member.

`ClassLength`

The size, in bytes, of the key class name string in the **Class** array.

`SubKeys`

The number of subkeys for this key.

`MaxNameLen`

The maximum size, in bytes, of any name for a subkey.

`MaxClassLen`

The maximum size, in bytes, of a class name.

`Values`

The number of value entries for this key.

`MaxValueNameLen`

The maximum size, in bytes, of a value entry name.

`MaxValueDataLen`

The maximum size, in bytes, of a value entry data field.

`Class`

An array of wide characters that contains the name of the class of the key. This character string is not null-terminated. Only the first element in this array is included in the **KEY_FULL_INFORMATION** structure definition. The storage for the remaining elements in the array immediately follows this element.

Remarks

The ZwEnumerateKey and ZwQueryKey routines use the **KEY_FULL_INFORMATION** structure to contain the full information for a registry key. When the *KeyInformationClass* parameter of either routine is **KeyFullInformation**, the *KeyInformation* buffer is treated as a **KEY_FULL_INFORMATION** structure. For more information about the **KeyFullInformation** enumeration value, see KEY_INFORMATION_CLASS.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwEnumerateKey

ZwQueryKey

KEY_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_INFORMATION_CLASS** enumeration type represents the type of information to supply about a registry key.

Syntax

```
typedef enum _KEY_INFORMATION_CLASS {
  KeyBasicInformation,
  KeyNodeInformation,
  KeyFullInformation,
  KeyNameInformation,
  KeyCachedInformation,
  KeyFlagsInformation,
  KeyVirtualizationInformation,
  KeyHandleTagsInformation,
  KeyTrustInformation,
  KeyLayerInformation,
  MaxKeyInfoClass
} KEY_INFORMATION_CLASS;
```

Constants

KeyBasicInformation	A KEY_BASIC_INFORMATION structure is supplied.
KeyNodeInformation	A KEY_NODE_INFORMATION structure is supplied.
KeyFullInformation	A KEY_FULL_INFORMATION structure is supplied.
KeyNameInformation	A KEY_NAME_INFORMATION structure is supplied.
KeyCachedInformation	A KEY_CACHED_INFORMATION structure is supplied.
KeyFlagsInformation	Reserved for system use.
KeyVirtualizationInformation	A KEY_VIRTUALIZATION_INFORMATION structure is supplied.
KeyHandleTagsInformation	Reserved for system use.
KeyTrustInformation	
KeyLayerInformation	
MaxKeyInfoClass	The maximum value in this enumeration type.

Remarks

Use the **KEY_INFORMATION_CLASS** values to specify the type of data to be supplied by the ZwEnumerateKey and ZwQueryKey routines.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_FULL_INFORMATION

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwEnumerateKey

ZwQueryKey

KEY_NODE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_NODE_INFORMATION** structure defines the basic information available for a registry (sub)key.

Syntax

```
typedef struct _KEY_NODE_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG         TitleIndex;
  ULONG         ClassOffset;
  ULONG         ClassLength;
  ULONG         NameLength;
  WCHAR         Name[1];
} KEY_NODE_INFORMATION, *PKEY_NODE_INFORMATION;
```

Members

`LastWriteTime`

The last time this key or any of its values changed. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

`TitleIndex`

Device and intermediate drivers should ignore this member.

`ClassOffset`

The byte offset from the start of this structure to the class name string, which is located in the **Name** array immediately following the key name string. Like the key name string, the class name string is not null-terminated.

`ClassLength`

The size, in bytes, in the class name string.

`NameLength`

The size, in bytes, of the key name string contained in the **Name** array.

`Name`

An array of wide characters that contains the name of the registry key. This character string is not null-terminated. Only the first element in this array is included in the **KEY_NODE_INFORMATION** structure definition. The storage for the remaining elements in the array immediately follows this element.

Remarks

The ZwEnumerateKey and ZwQueryKey routines use the **KEY_NODE_INFORMATION** structure to contain the registry key name and key class name. When the *KeyInformationClass* parameter of either routine is **KeyNodeInformation**, the *KeyInformation* buffer is treated as a **KEY_NODE_INFORMATION** structure. For more information about the **KeyNodeInformation** enumeration value, see KEY_INFORMATION_CLASS.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

ZwEnumerateKey

ZwQueryKey

KEY_SET_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_SET_INFORMATION_CLASS** enumeration type represents the type of information to set for a registry key.

Syntax

```
typedef enum _KEY_SET_INFORMATION_CLASS {
  KeyWriteTimeInformation,
  KeyWow64FlagsInformation,
  KeyControlFlagsInformation,
  KeySetVirtualizationInformation,
  KeySetDebugInformation,
  KeySetHandleTagsInformation,
  KeySetLayerInformation,
  MaxKeySetInfoClass
} KEY_SET_INFORMATION_CLASS;
```

Constants

KeyWriteTimeInformation	Indicates that a KEY_WRITE_TIME_INFORMATION structure is supplied.
KeyWow64FlagsInformation	Reserved for system use.
KeyControlFlagsInformation	Reserved for system use.
KeySetVirtualizationInformation	Reserved for system use.
KeySetDebugInformation	Reserved for system use.
KeySetHandleTagsInformation	Reserved for system use.
KeySetLayerInformation	
MaxKeySetInfoClass	This member constant is always the maximum value in the enumeration.

Remarks

A RegistryCallback routine can receive a pointer to a **KEY_SET_INFORMATION_CLASS** structure as an input parameter.

Requirements

Minimum supported client	Available starting with Windows XP.

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_WRITE_TIME_INFORMATION

RegistryCallback

KEY_VALUE_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VALUE_BASIC_INFORMATION** structure defines a subset of the full information available for a value entry of a registry key.

Syntax

```
typedef struct _KEY_VALUE_BASIC_INFORMATION {
  ULONG TitleIndex;
  ULONG Type;
  ULONG NameLength;
  WCHAR Name[1];
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

Members

`TitleIndex`

Device and intermediate drivers should ignore this member.

`Type`

Specifies the system-defined type for the value entry in the registry key, which is one of the following:

REG_*XXX* TYPE	**VALUE**
REG_BINARY	Binary data in any form
REG_DWORD	A 4-byte numerical value
REG_DWORD_LITTLE_ENDIAN	A 4-byte numerical value whose least significant byte is at the lowest address
REG_DWORD_BIG_ENDIAN	A 4-byte numerical value whose least significant byte is at the highest address
REG_EXPAND_SZ	A null-terminated Unicode string, containing unexpanded references to environment variables, such as "%PATH%"
REG_LINK	A Unicode string naming a symbolic link. This type is irrelevant to device and intermediate drivers
REG_MULTI_SZ	An array of null-terminated strings, terminated by another zero
REG_NONE	Data with no particular type
REG_SZ	A null-terminated Unicode string

REG_RESOURCE_LIST	A device driver's list of hardware resources, used by the driver or one of the physical devices it controls, in the **\ResourceMap** tree
REG_RESOURCE_REQUIREMENTS_LIST	A device driver's list of possible hardware resources it or one of the physical devices it controls can use, from which the system writes a subset into the **\ResourceMap** tree
REG_FULL_RESOURCE_DESCRIPTOR	A list of hardware resources that a physical device is using, detected and written into the **\HardwareDescription** tree by the system

`NameLength`

Specifies the size in bytes of the following value entry name.

`Name`

A string of Unicode characters naming a value entry of the key.

Remarks

A kernel-mode driver can obtain a **KEY_VALUE_BASIC_INFORMATION** that describes a registry key by calling the ZwQueryValueKey or ZwEnumerateValueKey routine.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_VALUE_FULL_INFORMATION

KEY_VALUE_INFORMATION_CLASS

KEY_VALUE_PARTIAL_INFORMATION

ZwEnumerateValueKey

ZwQueryValueKey

KEY_VALUE_ENTRY structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VALUE_ENTRY** structure is used by the REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure to describe a single value entry for a registry key.

Syntax

```
typedef struct _KEY_VALUE_ENTRY {
  PUNICODE_STRING ValueName;
  ULONG           DataLength;
  ULONG           DataOffset;
  ULONG           Type;
} KEY_VALUE_ENTRY, *PKEY_VALUE_ENTRY;
```

Members

`ValueName`

Pointer to a UNICODE_STRING structure that contains the name of the value entry.

`DataLength`

Specifies the size, in bytes, of the data for the value entry.

`DataOffset`

Specifies the offset, in bytes, of the value entry's data within the buffer that is pointed to by the **ValueBuffer** member of REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION.

`Type`

Specifies the type of the value entry's data. For a description of the possible values for **Type**, see the *Type* parameter of ZwSetValueKey.

Remarks

The **ValueEntries** member of REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION points to an array of KEY_VALUE_ENTRY structures.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION

ZwSetValueKey

KEY_VALUE_FULL_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VALUE_FULL_INFORMATION** structure defines information available for a value entry of a registry key.

Syntax

```
typedef struct _KEY_VALUE_FULL_INFORMATION {
  ULONG TitleIndex;
  ULONG Type;
  ULONG DataOffset;
  ULONG DataLength;
  ULONG NameLength;
  WCHAR Name[1];
} KEY_VALUE_FULL_INFORMATION, *PKEY_VALUE_FULL_INFORMATION;
```

Members

`TitleIndex`

Device and intermediate drivers should ignore this member.

`Type`

Specifies the system-defined type for the registry value(s) following the **Name** member. For a summary of these types, see KEY_VALUE_BASIC_INFORMATION.

`DataOffset`

Specifies the offset from the start of this structure to the data immediately following the **Name** string.

`DataLength`

Specifies the number of bytes of registry information for the value entry identified by **Name**.

`NameLength`

Specifies the size in bytes of the following value entry name.

`Name`

A string of Unicode characters naming a value entry of the key.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_VALUE_BASIC_INFORMATION

KEY_VALUE_INFORMATION_CLASS

KEY_VALUE_PARTIAL_INFORMATION

ZwEnumerateValueKey

ZwQueryValueKey

KEY_VALUE_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VALUE_INFORMATION_CLASS** enumeration type specifies the type of information to supply about the value of a registry key.

Syntax

```
typedef enum _KEY_VALUE_INFORMATION_CLASS {
  KeyValueBasicInformation,
  KeyValueFullInformation,
  KeyValuePartialInformation,
  KeyValueFullInformationAlign64,
  KeyValuePartialInformationAlign64,
  KeyValueLayerInformation,
  MaxKeyValueInfoClass
} KEY_VALUE_INFORMATION_CLASS;
```

Constants

KeyValueBasicInformation	The information is stored as a KEY_VALUE_BASIC_INFORMATION structure.
KeyValueFullInformation	The information is stored as a KEY_VALUE_FULL_INFORMATION structure.
KeyValuePartialInformation	The information is stored as a KEY_VALUE_PARTIAL_INFORMATION structure.
KeyValueFullInformationAlign64	The information is stored as a **KEY_VALUE_FULL_INFORMATION** structure that is aligned to a 64-bit (that is, 8-byte) boundary in memory. If the caller-supplied buffer does not start on a 64-bit boundary, the information is stored starting at the first 64-bit boundary in the buffer.
KeyValuePartialInformationAlign64	The information is stored as a **KEY_VALUE_PARTIAL_INFORMATION** structure that is aligned to a 64-bit (that is, 8-byte) boundary in memory. If the caller-supplied buffer does not start on a 64-bit boundary, the information is stored starting at the first 64-bit boundary in the buffer.
KeyValueLayerInformation	
MaxKeyValueInfoClass	The maximum value in this enumeration type.

Remarks

Use the **KEY_VALUE_INFORMATION_CLASS** values to specify the type of data to be supplied by the ZwEnumerateValueKey and ZwQueryValueKey routines.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_VALUE_BASIC_INFORMATION

KEY_VALUE_FULL_INFORMATION

KEY_VALUE_PARTIAL_INFORMATION

ZwEnumerateValueKey

ZwQueryValueKey

KEY_VALUE_PARTIAL_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_VALUE_PARTIAL_INFORMATION** structure defines a subset of the value information available for a value entry of a registry key.

Syntax

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
  ULONG TitleIndex;
  ULONG Type;
  ULONG DataLength;
  UCHAR Data[1];
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

Members

`TitleIndex`

Device and intermediate drivers should ignore this member.

`Type`

Specifies the system-defined type for the registry value in the **Data** member. For a summary of these types, see KEY_VALUE_BASIC_INFORMATION.

`DataLength`

The size in bytes of the **Data** member.

`Data`

A value entry of the key.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_VALUE_BASIC_INFORMATION

KEY_VALUE_FULL_INFORMATION

KEY_VALUE_INFORMATION_CLASS

ZwEnumerateValueKey

ZwQueryValueKey

KEY_WRITE_TIME_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **KEY_WRITE_TIME_INFORMATION** structure is used by the system to set the last write time for a registry key.

Syntax

```
typedef struct _KEY_WRITE_TIME_INFORMATION {
  LARGE_INTEGER LastWriteTime;
} KEY_WRITE_TIME_INFORMATION, *PKEY_WRITE_TIME_INFORMATION;
```

Members

`LastWriteTime`

Specifies the last time that the key was changed. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

KEY_SET_INFORMATION_CLASS

KINTERRUPT_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KINTERRUPT_MODE** enumeration type indicates whether an interrupt is level-triggered or edge-triggered.

Syntax

```
typedef enum _KINTERRUPT_MODE {
  LevelSensitive,
  Latched
} KINTERRUPT_MODE;
```

Constants

LevelSensitive	The interrupt is level-triggered. This is the mode for traditional PCI line-based interrupts.
Latched	The interrupt is edge-triggered. This is the mode for PCI message-signaled interrupts.

Remarks

The interrupt mode is an electrical characteristic of the interrupt. A device signals a level-triggered interrupt by changing the voltage on the interrupt pin, and holding it there until the processor signals that the interrupt is received. In contrast, to signal an edge-triggered interrupt, the device changes the voltage state for a fixed period of time, before reverting to the original voltage state.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

IoConnectInterrupt

IoConnectInterruptEx

KINTERRUPT_POLARITY

KINTERRUPT_POLARITY Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KINTERRUPT_POLARITY** enumeration indicates how a device signals an interrupt request on an interrupt line.

Syntax

```
typedef enum _KINTERRUPT_POLARITY {
  InterruptPolarityUnknown,
  InterruptActiveHigh,
  InterruptRisingEdge,
  InterruptActiveLow,
  InterruptFallingEdge,
  InterruptActiveBoth,
  InterruptActiveBothTriggerLow,
  InterruptActiveBothTriggerHigh
} KINTERRUPT_POLARITY, *PKINTERRUPT_POLARITY;
```

Constants

InterruptPolarityUnknown	The interrupt polarity is unknown.
InterruptActiveHigh	Active-high interrupt. The interrupt input type is level-triggered, and an interrupt request is indicated by a high signal level on the interrupt line. The request remains active as long as the line remains high.
InterruptRisingEdge	Rising-edge-triggered interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a low-to-high transition on the interrupt line.
InterruptActiveLow	Active-low interrupt. The interrupt input type is level-triggered, and an interrupt request is indicated by a low signal level on the interrupt line. The request remains active as long as the line remains low.
InterruptFallingEdge	Falling-edge-triggered interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a high-to-low transition on the interrupt line.
InterruptActiveBoth	Active-both interrupt. The interrupt input type is edge-triggered, and an interrupt request is indicated by a low-to-high or a high-to-low transition on the interrupt line. After a low-to-high transition signals an interrupt request, the interrupt line remains high until a high-to-low transition signals the next interrupt request. Similarly, after a high-to-low transition signals an interrupt request, the interrupt line remains low until a low-to-high transition signals the next interrupt request.
InterruptActiveBothTriggerLow	Reserved for use by the operating system.

InterruptActiveBothTriggerHigh	Reserved for use by the operating system.

Remarks

A **KINTERRUPT_POLARITY** enumeration constant is frequently used in conjunction with a KINTERRUPT_MODE enumeration constant to describe an interrupt signal. A **KINTERRUPT_MODE** enumeration constant indicates whether the interrupt signal from a device is level-triggered or edge-triggered.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

KINTERRUPT_MODE

KTMOBJECT_CURSOR structure

1/30/2019 • 2 minutes to read • Edit Online

The **KTMOBJECT_CURSOR** structure receives enumeration information about KTM objects when a component calls ZwEnumerateTransactionObject.

Syntax

```
typedef struct _KTMOBJECT_CURSOR {
  GUID  LastQuery;
  ULONG ObjectIdCount;
  GUID  ObjectIds[1];
} KTMOBJECT_CURSOR, *PKTMOBJECT_CURSOR;
```

Members

`LastQuery`

After **ZwEnumerateTransactionObject** returns, this member contains the GUID of the last object that **ZwEnumerateTransactionObject** enumerated. Before it calls **ZwEnumerateTransactionObject** the first time, the caller must set this value to zero.

`ObjectIdCount`

After **ZwEnumerateTransactionObject** returns, this member contains the number of GUIDs that the **ObjectIds** array contains.

`ObjectIds`

A caller-allocated array of GUID-typed elements. After **ZwEnumerateTransactionObject** returns, this array contains GUIDs that identify enumerated objects.

Remarks

The **KTMOBJECT_CURSOR** structure is used at the beginning of buffers that callers pass to the **ZwEnumerateTransactionObject** routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwEnumerateTransactionObject

KTMOBJECT_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KTMOBJECT_TYPE** enumeration identifies the types of objects that KTM supports.

Syntax

```
typedef enum _KTMOBJECT_TYPE {
  KTMOBJECT_TRANSACTION,
  KTMOBJECT_TRANSACTION_MANAGER,
  KTMOBJECT_RESOURCE_MANAGER,
  KTMOBJECT_ENLISTMENT,
  KTMOBJECT_INVALID
} KTMOBJECT_TYPE, *PKTMOBJECT_TYPE;
```

Constants

KTMOBJECT_TRANSACTION	KTM transaction objects.
KTMOBJECT_TRANSACTION_MANAGER	KTM transaction manager objects.
KTMOBJECT_RESOURCE_MANAGER	KTM resource manager objects.
KTMOBJECT_ENLISTMENT	KTM enlistment objects.
KTMOBJECT_INVALID	Invalid object type.

Remarks

The **KTMOBJECT_TYPE** enumeration is used with the ZwEnumerateTransactionObject routine.

For more information about KTM objects, see KTM Objects.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwEnumerateTransactionObject

LINK_SHARE_ACCESS structure

1/30/2019 • 2 minutes to read • Edit Online

The share access structure used by file systems for only link files.

Syntax

```
typedef struct _LINK_SHARE_ACCESS {
  ULONG OpenCount;
  ULONG Deleters;
  ULONG SharedDelete;
} LINK_SHARE_ACCESS, *PLINK_SHARE_ACCESS;
```

Members

`OpenCount`

The number of open requests to the file.

`Deleters`

File associated with the file object has been opened for delete access.

`SharedDelete`

File associated with the file object has been opened for delete sharing access.

Requirements

Header	wdm.h

See Also

IoCheckLinkShareAccess

IoSetLinkShareAccess

MDL structure

1/30/2019 • 2 minutes to read • Edit Online

An **MDL** structure is a partially opaque structure that represents a memory descriptor list (MDL).

Syntax

```
typedef struct _MDL {
  struct _MDL      *Next;
  CSHORT           Size;
  CSHORT           MdlFlags;
  struct _EPROCESS *Process;
  PVOID            MappedSystemVa;
  PVOID            StartVa;
  ULONG            ByteCount;
  ULONG            ByteOffset;
} MDL, *PMDL;
```

Members

`Next`

Pointer to the next MDL in an MDL chain. For more information about MDL chains, see Using MDLs.

`Size`

`MdlFlags`

`Process`

`MappedSystemVa`

`StartVa`

`ByteCount`

`ByteOffset`

Remarks

An MDL describes the layout of a virtual memory buffer in physical memory. For more information, see Using MDLs.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoAllocateMdl

MmGetMdlByteCount

MmGetMdlByteOffset

MmGetMdlPfnArray

MmGetMdlVirtualAddress

MmInitializeMdl

MEMORY_CACHING_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **MEMORY_CACHING_TYPE** enumeration type specifies the permitted caching behavior when allocating or mapping memory.

Syntax

```
typedef enum _MEMORY_CACHING_TYPE {
  MmNonCached,
  MmCached,
  MmWriteCombined,
  MmHardwareCoherentCached,
  MmNonCachedUnordered,
  MmUSWCCached,
  MmMaximumCacheType,
  MmNotMapped
} MEMORY_CACHING_TYPE;
```

Constants

MmNonCached	The requested memory should not be cached by the processor.
MmCached	The processor should cache the requested memory.
MmWriteCombined	The requested memory should not be cached by the processor, but writes to the memory can be combined by the processor.
MmHardwareCoherentCached	Reserved for system use.
MmNonCachedUnordered	Reserved for system use.
MmUSWCCached	Reserved for system use.
MmMaximumCacheType	Reserved for system use.
MmNotMapped	

Remarks

Processor translation buffers cache virtual to physical address translations. These translation buffers allow many virtual addresses to map a single physical address. However, only one caching behavior is allowed for any given physical address translation. Therefore, if a driver maps two different virtual address ranges to the same physical address, it must ensure that it specifies the same caching behavior for both. Otherwise, the processor behavior is undefined with unpredictable system results.

Requirements

Minimum supported client	Supported in Windows 2000 and later versions of Windows.
Header	wdm.h (include Ntddk.h, Wdm.h, Miniport.h, Wudfwdm.h)

See Also

MmAllocateContiguousMemorySpecifyCache

MmFreeContiguousMemorySpecifyCache

MmMapIoSpace

MmMapLockedPagesSpecifyCache

MmMapLockedPagesWithReservedMapping

MM_PHYSICAL_ADDRESS_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **MM_PHYSICAL_ADDRESS_LIST** structure specifies a range of physical addresses.

Syntax

```
typedef struct _MM_PHYSICAL_ADDRESS_LIST {
  PHYSICAL_ADDRESS PhysicalAddress;
  SIZE_T           NumberOfBytes;
} MM_PHYSICAL_ADDRESS_LIST, *PMM_PHYSICAL_ADDRESS_LIST;
```

Members

`PhysicalAddress`

The starting physical address of the range. This address must be aligned to a page boundary in physical memory.

`NumberOfBytes`

The number of bytes in the range. This member must be nonzero and an integer multiple of the memory page size.

Remarks

The first parameter to the MmAllocateMdlForIoSpace routine is a pointer to an array of **MM_PHYSICAL_ADDRESS_LIST** structures.

Requirements

Header	wdm.h (include Wdm.h)

See Also

MmAllocateMdlForIoSpace

MONITOR_DISPLAY_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

Indicates the power state of the monitor being displayed on.

Syntax

```
typedef enum _MONITOR_DISPLAY_STATE {
  PowerMonitorOff,
  PowerMonitorOn,
  PowerMonitorDim
} MONITOR_DISPLAY_STATE, *PMONITOR_DISPLAY_STATE;
```

Constants

PowerMonitorOff	This indicates that the monitor is off.
PowerMonitorOn	This indicates that the monitor is on.
PowerMonitorDim	This indicates that the monitor is dim.

Requirements

Header	wdm.h (include Ntpoapi.h)

OB_CALLBACK_REGISTRATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_CALLBACK_REGISTRATION** structure specifies the parameters when the ObRegisterCallbacks routine registers ObjectPreCallback and ObjectPostCallback callback routines.

Syntax

```
typedef struct _OB_CALLBACK_REGISTRATION {
  USHORT                    Version;
  USHORT                    OperationRegistrationCount;
  UNICODE_STRING            Altitude;
  PVOID                     RegistrationContext;
  OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

Members

`Version`

The version of object callback registration that is requested. Drivers should specify OB_FLT_REGISTRATION_VERSION.

`OperationRegistrationCount`

The number of entries in the **OperationRegistration** array.

`Altitude`

A Unicode string that specifies the altitude of the driver. For more information about altitude, see Load Order Groups and Altitudes for Minifilter Drivers.

`RegistrationContext`

The system passes the **RegistrationContext** value to the callback routine when the callback routine is run. The meaning of this value is driver-defined.

`OperationRegistration`

A pointer to an array of OB_OPERATION_REGISTRATION structures. Each structure specifies ObjectPreCallback and ObjectPostCallback callback routines and the types of operations that the routines are called for.

Remarks

This structure is used by the ObRegisterCallbacks routine. The *CallBackRegistration* parameter to this routine is a pointer to a buffer that contains an **OB_CALLBACK_REGISTRATION** structure that is followed by an array of one or more OB_OPERATION_REGISTRATION structures.

Requirements

Minimum supported client	Available starting with Windows Server 2008.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_OPERATION_REGISTRATION

ObRegisterCallbacks

ObjectPostCallback

ObjectPreCallback

OB_OPERATION_REGISTRATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_OPERATION_REGISTRATION** structure specifies ObjectPreCallback and ObjectPostCallback callback routines and the types of operations that the routines are called for.

Syntax

```
typedef struct _OB_OPERATION_REGISTRATION {
  POBJECT_TYPE                *ObjectType;
  OB_OPERATION                Operations;
  POB_PRE_OPERATION_CALLBACK  PreOperation;
  POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

Members

`ObjectType`

A pointer to the object type that triggers the callback routine. Specify one of the following values:

- **PsProcessType** for process handle operations
- **PsThreadType** for thread handle operations
- **ExDesktopObjectType** for desktop handle operations. This value is supported in Windows 10 and not in the earlier versions of the operating system.

`Operations`

Specify one or more of the following flags:

OB_OPERATION_HANDLE_CREATE

A new process, thread, or desktop handle was or will be opened.

OB_OPERATION_HANDLE_DUPLICATE

A process, thread, or desktop handle was or will be duplicated.

`PreOperation`

A pointer to an ObjectPreCallback routine. The system calls this routine before the requested operation occurs.

`PostOperation`

A pointer to an ObjectPostCallback routine. The system calls this routine after the requested operation occurs.

Remarks

This structure is used by the ObRegisterCallbacks routine. The *CallBackRegistration* parameter to this routine is a pointer to a buffer that contains an OB_CALLBACK_REGISTRATION structure that is followed by an array of one or more **OB_OPERATION_REGISTRATION** structures.

In each **OB_OPERATION_REGISTRATION** structure passed to **ObRegisterCallback**, the caller must supply one or both callback routines. If the **PreOperation** and **PostOperation** members of this structure are both **NULL**, the callback registration operation fails.

Requirements

Minimum supported client	Available starting with Windows Server 2008.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_CALLBACK_REGISTRATION

ObRegisterCallbacks

ObjectPostCallback

ObjectPreCallback

OB_POST_CREATE_HANDLE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_POST_CREATE_HANDLE_INFORMATION** structure provides information to a ObjectPostCallback routine about a thread or process handle that has been opened.

Syntax

```
typedef struct _OB_POST_CREATE_HANDLE_INFORMATION {
  ACCESS_MASK GrantedAccess;
} OB_POST_CREATE_HANDLE_INFORMATION, *POB_POST_CREATE_HANDLE_INFORMATION;
```

Members

`GrantedAccess`

An ACCESS_MASK value that specifies the access that is granted for the handle.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

ObjectPostCallback

OB_POST_DUPLICATE_HANDLE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_POST_DUPLICATE_HANDLE_INFORMATION** structure provides information to an ObjectPostCallback routine about a thread or process handle that has been duplicated.

Syntax

```
typedef struct _OB_POST_DUPLICATE_HANDLE_INFORMATION {
  ACCESS_MASK GrantedAccess;
} OB_POST_DUPLICATE_HANDLE_INFORMATION, *POB_POST_DUPLICATE_HANDLE_INFORMATION;
```

Members

`GrantedAccess`

An ACCESS_MASK value that specifies the access that is granted for the handle.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

ObjectPostCallback

OB_POST_OPERATION_INFORMATION structure

2/5/2019 • 2 minutes to read • Edit Online

The **OB_POST_OPERATION_INFORMATION** structure provides information about a process or thread handle operation to an ObjectPostCallback routine.

Syntax

```
typedef struct _OB_POST_OPERATION_INFORMATION {
  OB_OPERATION                  Operation;
  union {
    ULONG Flags;
    struct {
      ULONG KernelHandle : 1;
      ULONG Reserved : 31;
    };
  };
  PVOID                         Object;
  POBJECT_TYPE                  ObjectType;
  PVOID                         CallContext;
  NTSTATUS                      ReturnStatus;
  POB_POST_OPERATION_PARAMETERS Parameters;
} OB_POST_OPERATION_INFORMATION, *POB_POST_OPERATION_INFORMATION;
```

Members

`Operation`

The type of handle operation. This member might be one of the following values:

OB_OPERATION_HANDLE_CREATE

A new handle to a process or thread was created. Use **Parameters->CreateHandleInformation** for create-specific information.

OB_OPERATION_HANDLE_DUPLICATE

A process or thread handle was duplicated. Use **Parameters->DuplicateHandleInformation** for duplicate-specific information.

`Flags`

Reserved. Use the **KernelHandle** member instead.

`KernelHandle`

A ULONG value that specifies whether the handle is a kernel handle. If this value is **TRUE**, the handle is a kernel handle. Otherwise, the handle is not a kernel handle.

`Reserved`

Reserved for system use.

`Object`

A pointer to the process or thread object that is the target of the handle operation.

`ObjectType`

A pointer to the object type of the object. This type can be **PsProcessType** for a process or **PsThreadType** for a thread.

`CallContext`

A pointer to driver-specific context information for the operation. This value is the value that the *OperationInformation*->**CallContext** member specifies to the ObjectPreCallback routine.

`ReturnStatus`

The NTSTATUS value for the handle operation.

`Parameters`

A pointer to an OB_POST_OPERATION_PARAMETERS union that contains operation-specific information. The **Operation** member determines which member of the union is valid. The pointer is valid only when **ReturnStatus** is a success code.

Remarks

Unlike the OB_PRE_OPERATION_INFORMATION structure, the members of the **OB_POST_OPERATION_INFORMATION** structure are purely informational; you cannot modify them.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_PRE_OPERATION_INFORMATION

ObjectPostCallback

ObjectPreCallback

OB_POST_OPERATION_PARAMETERS union

1/30/2019 • 2 minutes to read • Edit Online

The **OB_POST_OPERATION_PARAMETERS** union describes the operation-specific parameters for an ObjectPostCallback routine.

Syntax

```
typedef union _OB_POST_OPERATION_PARAMETERS {
  OB_POST_CREATE_HANDLE_INFORMATION    CreateHandleInformation;
  OB_POST_DUPLICATE_HANDLE_INFORMATION DuplicateHandleInformation;
} OB_POST_OPERATION_PARAMETERS, *POB_POST_OPERATION_PARAMETERS;
```

Members

`CreateHandleInformation`

An OB_POST_CREATE_HANDLE_INFORMATION structure that contains information that is specific to a handle that is being opened.

`DuplicateHandleInformation`

An OB_POST_DUPLICATE_HANDLE_INFORMATION structure that contains information that is specific to a handle that is being duplicated.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_POST_CREATE_HANDLE_INFORMATION

OB_POST_DUPLICATE_HANDLE_INFORMATION

ObjectPostCallback

OB_PRE_CREATE_HANDLE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_PRE_CREATE_HANDLE_INFORMATION** structure provides information to an ObjectPreCallback routine about a thread or process handle that is being opened.

Syntax

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
  ACCESS_MASK DesiredAccess;
  ACCESS_MASK OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;
```

Members

`DesiredAccess`

An ACCESS_MASK value that specifies the access rights to grant for the handle. By default, this member equals *OriginalDesiredAccess*, but the *ObjectPreCallback* routine can modify this value to restrict the access that is granted.

Drivers can use the following flags for handles to processes:

FLAG	ALLOWED OPERATIONS
PROCESS_CREATE_PROCESS	Create a new child process of the process.
PROCESS_CREATE_THREAD	Create a new thread in the context of the process.
PROCESS_DUP_HANDLE	Duplicate handles to or from the context of the process, such as by calling the user-mode **DuplicateHandle** routine.
PROCESS_SET_QUOTA	Set the working set size for the process, such as by calling the user-mode **SetProcessWorkingSetSize** routine.
PROCESS_SET_INFORMATION	Modify process settings, such as by calling the user-mode **SetPriorityClass** routine.
PROCESS_SUSPEND_RESUME	Suspend or resume the process.
PROCESS_TERMINATE	Terminate the process, such as by calling the user-mode **TerminateProcess** routine..
PROCESS_VM_OPERATION	Modify the address space of the process, such as by calling the user-mode **WriteProcessMemory** and **VirtualProtectEx** routines.
PROCESS_VM_WRITE	Write to the address space of the process, such as by calling the user-mode **WriteProcessMemory** routine.

Drivers can use the following flags for handles to threads:

FLAG	ALLOWED OPERATIONS
THREAD_DIRECT_IMPERSONATION	Enable a server thread to impersonate one of its clients.
THREAD_IMPERSONATE	Impersonate the operating system's anonymous logon token, such as by calling the user-mode **ImpersonateAnonymousToken** routine.
THREAD_SET_CONTEXT	Modify the thread's execution context, such as by calling the user-mode **SetThreadContext** routine.
THREAD_SET_INFORMATION	Modify thread settings, such as by calling the user-mode **SetThreadIdealProcessor** routine. The operations that are permitted by this access right are a superset of those that are permitted by the THREAD_SET_LIMITED_INFORMATION access right.
THREAD_SET_LIMITED_INFORMATION	Modify a limited set of thread settings, such as by calling the user-mode **SetThreadAffinityMask** and **SetThreadPriorityBoost** routines.
THREAD_SET_THREAD_TOKEN	Modify properties of the thread's impersonation token, such as by calling the user-mode **SetTokenInformation** routine.
THREAD_SUSPEND_RESUME	Suspend or resume the thread, such as by calling the user-mode **SuspendThread** and **ResumeThread** routines.
THREAD_TERMINATE	Terminate the thread, such as by calling the user-mode **TerminateThread** routine.

`OriginalDesiredAccess`

An ACCESS_MASK value that specifies the original access that was requested for the handle.

Remarks

You can never add access rights beyond what is specified in the **DesiredAccess** member. If the access right is listed as a modifiable flag, the access right can be removed.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

ObjectPreCallback

OB_PRE_DUPLICATE_HANDLE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **OB_PRE_DUPLICATE_HANDLE_INFORMATION** structure provides information to an ObjectPreCallback routine about a thread or process handle that is being duplicated.

Syntax

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
  ACCESS_MASK DesiredAccess;
  ACCESS_MASK OriginalDesiredAccess;
  PVOID       SourceProcess;
  PVOID       TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, *POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

Members

`DesiredAccess`

An ACCESS_MASK value that specifies the access rights to grant for the handle. By default, this member equals *OriginalDesiredAccess*, but the *ObjectPreCallback* routine can modify this value to restrict the access that is granted. For a description of the access rights that drivers can use, see OB_PRE_CREATE_HANDLE_INFORMATION.

`OriginalDesiredAccess`

An ACCESS_MASK value that specifies the original access that was requested for the handle.

`SourceProcess`

A pointer to the process object for the process that is the source of the handle.

`TargetProcess`

A pointer to the process object for the process that receives the duplicated handle.

Remarks

You can never add access rights beyond what is specified in the **DesiredAccess** member. If the access right is listed as a modifiable flag, the access right can be removed.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ACCESS_MASK

ObjectPreCallback

OB_PRE_OPERATION_INFORMATION structure

2/5/2019 • 2 minutes to read • Edit Online

The **OB_PRE_OPERATION_INFORMATION** structure provides information about a process or thread handle operation to an ObjectPreCallback routine.

Syntax

```
typedef struct _OB_PRE_OPERATION_INFORMATION {
  OB_OPERATION                  Operation;
  union {
    ULONG Flags;
    struct {
      ULONG KernelHandle : 1;
      ULONG Reserved : 31;
    };
  };
  PVOID                         Object;
  POBJECT_TYPE                  ObjectType;
  PVOID                         CallContext;
  POB_PRE_OPERATION_PARAMETERS Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;
```

Members

`Operation`

The type of handle operation. This member might be one of the following values:

OB_OPERATION_HANDLE_CREATE

A new handle to a process or thread will be opened. Use **Parameters->CreateHandleInformation** for create-specific information.

OB_OPERATION_HANDLE_DUPLICATE

A process or thread handle will be duplicated. Use **Parameters->DuplicateHandleInformation** for duplicate-specific information.

`Flags`

Reserved. Use the **KernelHandle** member instead.

`KernelHandle`

A bit that specifies whether the handle is a kernel handle. If this member is **TRUE**, the handle is a kernel handle. Otherwise, this handle is not a kernel handle.

`Reserved`

Reserved for system use.

`Object`

A pointer to the process or thread object that is the target of the handle operation.

`ObjectType`

A pointer to the object type of the object. This member is **PsProcessType** for a process or **PsThreadType** for a thread.

`CallContext`

A pointer to driver-specific context information for the operation. By default, the Filter Manager sets this member to **NULL**, but the ObjectPreCallback routine can reset the **CallContext** member in a driver-specific manner. The Filter Manager passes this value to the matching ObjectPostCallback routine.

`Parameters`

A pointer to an OB_PRE_OPERATION_PARAMETERS union that contains operation-specific information. The **Operation** member determines which member of the union is valid.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_PRE_OPERATION_PARAMETERS

ObjectPostCallback

ObjectPreCallback

OB_PRE_OPERATION_PARAMETERS union

1/30/2019 • 2 minutes to read • Edit Online

The **OB_PRE_OPERATION_PARAMETERS** union describes the operation-specific parameters for an ObjectPreCallback routine.

Syntax

```
typedef union _OB_PRE_OPERATION_PARAMETERS {
  OB_PRE_CREATE_HANDLE_INFORMATION    CreateHandleInformation;
  OB_PRE_DUPLICATE_HANDLE_INFORMATION DuplicateHandleInformation;
} OB_PRE_OPERATION_PARAMETERS, *POB_PRE_OPERATION_PARAMETERS;
```

Members

`CreateHandleInformation`

An OB_PRE_CREATE_HANDLE_INFORMATION structure that contains information that is specific to a handle that is being opened.

`DuplicateHandleInformation`

An OB_PRE_DUPLICATE_HANDLE_INFORMATION structure that contains information that is specific to a handle that is being duplicated.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

OB_PRE_CREATE_HANDLE_INFORMATION

OB_PRE_DUPLICATE_HANDLE_INFORMATION

ObjectPreCallback

OSVERSIONINFOEXW structure

1/30/2019 • 4 minutes to read • Edit Online

The **RTL_OSVERSIONINFOEXW** structure contains operating system version information.

Syntax

```
typedef struct _OSVERSIONINFOEXW {
  ULONG  dwOSVersionInfoSize;
  ULONG  dwMajorVersion;
  ULONG  dwMinorVersion;
  ULONG  dwBuildNumber;
  ULONG  dwPlatformId;
  WCHAR  szCSDVersion[128];
  USHORT wServicePackMajor;
  USHORT wServicePackMinor;
  USHORT wSuiteMask;
  UCHAR  wProductType;
  UCHAR  wReserved;
} OSVERSIONINFOEXW, *POSVERSIONINFOEXW, *LPOSVERSIONINFOEXW, RTL_OSVERSIONINFOEXW, *PRTL_OSVERSIONINFOEXW;
```

Members

`dwOSVersionInfoSize`

The size, in bytes, of an **RTL_OSVERSIONINFOEXW** structure. This member must be set before the structure is used with RtlGetVersion.

`dwMajorVersion`

The major version number of the operating system. For example, for Windows 2000, the major version number is five. For more information, see the table in Remarks.

`dwMinorVersion`

The minor version number of the operating system. For example, for Windows 2000, the minor version number is zero. For more information, see the table in Remarks.

`dwBuildNumber`

The build number of the operating system.

`dwPlatformId`

The operating system platform. For Win32 on NT-based operating systems, **RtlGetVersion** returns the value VER_PLATFORM_WIN32_NT.

`szCSDVersion`

The service-pack version string. This member contains a null-terminated string, such as "Service Pack 3", which indicates the latest service pack installed on the system. If no service pack is installed, **RtlGetVersion** might not initialize this string. Initialize *szCSDVersion* to zero (empty string) before the call to **RtlGetVersion**.

`wServicePackMajor`

The major version number of the latest service pack installed on the system. For example, for Service Pack 3, the

major version number is three. If no service pack has been installed, the value is zero.

`wServicePackMinor`

The minor version number of the latest service pack installed on the system. For example, for Service Pack 3, the minor version number is zero.

`wSuiteMask`

The product suites available on the system. This member is set to zero or to the bitwise OR of one or more of the following values.

VALUE	MEANING
VER_SUITE_BACKOFFICE	Microsoft BackOffice components are installed.
VER_SUITE_BLADE	Windows Server 2003, Web Edition is installed.
VER_SUITE_COMPUTE_SERVER	Windows Server 2003, Compute Cluster Edition is installed.
VER_SUITE_DATACENTER	Windows Server 2008 Datacenter, Windows Server 2003, Datacenter Edition, or Windows 2000 Datacenter Server is installed.
VER_SUITE_ENTERPRISE	Windows Server 2008 Enterprise, Windows Server 2003, Enterprise Edition, or Windows 2000 Advanced Server is installed.
VER_SUITE_EMBEDDEDNT	Windows XP Embedded is installed.
VER_SUITE_PERSONAL	Windows Vista Home Premium, Windows Vista Home Basic, or Windows XP Home Edition is installed.
VER_SUITE_SINGLEUSERTS	Remote Desktop is supported, but only one interactive session is supported. This value is set unless the system is running in application server mode.
VER_SUITE_SMALLBUSINESS (see note)	Microsoft Small Business Server was once installed on the system, but may have been upgraded to another version of Windows. For more information about this flag bit, see the following Remarks section.
VER_SUITE_SMALLBUSINESS_RESTRICTED	Microsoft Small Business Server is installed with the restrictive client license in force. For more information about this flag bit, see the following Remarks section.
VER_SUITE_STORAGE_SERVER	Windows Storage Server 2003 R2 or Windows Storage Server 2003 is installed.
VER_SUITE_TERMINAL	Terminal Services is installed. This value is always set. If VER_SUITE_TERMINAL is set but VER_SUITE_SINGLEUSERTS is not set, the operating system is running in application server mode.
VER_SUITE_WH_SERVER	Windows Home Server is installed.

Note You should not rely solely on the VER_SUITE_SMALLBUSINESS flag to determine whether Small Business Server is currently installed. Both this flag and the VER_SUITE_SMALLBUSINESS_RESTRICTED flag are set when this product suite is installed. If you upgrade this installation to Windows Server, Standard Edition, the VER_SUITE_SMALLBUSINESS_RESTRICTED flag is cleared, but the VER_SUITE_SMALLBUSINESS flag remains set, which, in this case, indicates that Small Business Server was previously installed on this system. If this installation is further upgraded to Windows Server, Enterprise Edition, the VER_SUITE_SMALLBUSINESS flag remains set.

`wProductType`

The product type. This member contains additional information about the system. This member can be one of the following values:

VALUE	MEANING
VER_NT_WORKSTATION	Windows 2000 or later professional version
VER_NT_DOMAIN_CONTROLLER	Windows 2000 or later domain controller
VER_NT_SERVER	Windows 2000 or later server

`wReserved`

Reserved for future use.

Remarks

The information in this structure includes the major and minor version numbers, the build number, the platform identifier, the installed product suites, and the latest service pack that is installed on the system. This structure is used with the RtlGetVersion and RtlVerifyVersionInfo routines.

Relying on version information is not always the best way to test whether a feature is available. For guidance, refer to the documentation for the feature you are interested in.

If possible, design the version detection code in your driver to enable the driver to run on future versions of Windows. If your driver requires a particular operating system version, be sure to treat this version as the minimum supported version, and not as the only version on which the driver can run.

The following table summarizes the version information that is returned by supported versions of Windows. Use the information in the "Other" column to distinguish between operating systems with identical version numbers.

OPERATING SYSTEM	VERSION NUMBER	DWMAJORVERSION	DWMINORVERSION	OTHER
Windows 10	10.0	10	0	**wProductType** == VER_NT_WORKSTATION
Windows 8.1	6.3	6	3	**wProductType** == VER_NT_WORKSTATION
Windows 8	6.2	6	2	**wProductType** == VER_NT_WORKSTATION

Windows Server 2012	6.2	6	2	**wProductType** != VER_NT_WORKSTATION
Windows 7	6.1	6	1	**wProductType** == VER_NT_WORKSTATION
Windows Server 2008 R2	6.1	6	1	**wProductType** != VER_NT_WORKSTATION
Windows Server 2008	6.0	6	0	**wProductType** != VER_NT_WORKSTATION
Windows Vista	6.0	6	0	**wProductType** == VER_NT_WORKSTATION
Windows Home Server	5.2	5	2	**wSuiteMask** == VER_SUITE_WH_SERVER
Windows Server 2003	5.2	5	2	Not applicable
Windows XP Professional x64 Edition (see note)	5.2	5	2	**wProductType** == VER_NT_WORKSTATION
Windows XP	5.1	5	1	Not applicable
Windows 2000	5.0	5	0	Not applicable

Note Only a 64-bit kernel-mode driver can run on Windows XP Professional x64 Edition. Therefore, a 32-bit kernel-mode driver can safely omit checking for this version of Windows.

Requirements

Header	wdm.h (include Ntddk.h)

See Also

RTL_OSVERSIONINFOW

RtlGetVersion

RtlVerifyVersionInfo

OSVERSIONINFOW structure

1/30/2019 • 2 minutes to read • Edit Online

The **RTL_OSVERSIONINFOW** structure contains operating system version information. The information includes major and minor version numbers, a build number, a platform identifier, and descriptive text about the operating system. The **RTL_OSVERSIONINFOW** structure is used with RtlGetVersion.

Syntax

```
typedef struct _OSVERSIONINFOW {
  ULONG dwOSVersionInfoSize;
  ULONG dwMajorVersion;
  ULONG dwMinorVersion;
  ULONG dwBuildNumber;
  ULONG dwPlatformId;
  WCHAR szCSDVersion[128];
} OSVERSIONINFOW, *POSVERSIONINFOW, *LPOSVERSIONINFOW, RTL_OSVERSIONINFOW, *PRTL_OSVERSIONINFOW;
```

Members

`dwOSVersionInfoSize`

The size in bytes of an **RTL_OSVERSIONINFOW** structure. This member must be set before the structure is used with RtlGetVersion.

`dwMajorVersion`

The major version number of the operating system. For example, for Windows 2000, the major version number is five. For more information, see RTL_OSVERSIONINFOEXW.

`dwMinorVersion`

The minor version number of the operating system. For example, for Windows 2000 the minor version number is zero. For more information, see RTL_OSVERSIONINFOEXW.

`dwBuildNumber`

The build number of the operating system.

`dwPlatformId`

The operating system platform. For Microsoft Win32 on NT-based operating systems, **RtlGetVersion** returns the value VER_PLATFORM_WIN32_NT.

`szCSDVersion`

The service-pack version string. This member contains a null-terminated string, such as "Service Pack 3", which indicates the latest service pack installed on the system. If no service pack is installed, **RtlGetVersion** might not initialize this string. Initialize *szCSDVersion* to zero (empty string) before the call to **RtlGetVersion**.

Remarks

For a list of the major and minor version numbers for the various versions of Windows, see RTL_OSVERSIONINFOEXW.

Requirements

Header	wdm.h (include Ntddk.h)

See Also

RTL_OSVERSIONINFOEXW

RtlGetVersion

RtlVerifyVersionInfo

PCI_COMMON_CONFIG structure

1/30/2019 • 2 minutes to read • Edit Online

The **PCI_COMMON_CONFIG** structure is obsolete. It defines standard PCI configuration information returned by the obsolete HalGetBusData or HalGetBusDataByOffset routine for the input *BusDataType* PCIConfiguration, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Syntax

```
typedef struct _PCI_COMMON_CONFIG {
  UCHAR      DeviceSpecific[192];
  base_class PCI_COMMON_HEADER;
} PCI_COMMON_CONFIG, *PPCI_COMMON_CONFIG;
```

Members

`DeviceSpecific`

Contains any device-specific initialization information that is available.

`PCI_COMMON_HEADER`

Remarks

Certain members of this structure have read-only values, so attempts to reset them are ignored. These members include the following: **VendorID**, **DeviceID**, **RevisionID**, **ProgIf**, **SubClass**, **BaseClass**, **HeaderType**, **InterruptPin**, **MinimumGrant**, and **MaximumLatency.**

Other members are provisionally read-only: that is, the system initializes them to their correct values, so drivers can safely treat them as read-only. However, they can be reset if a bus-master driver finds it necessary. These members include the following: **CacheLineSize** and **LatencyTimer**.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

PCI_MSIX_TABLE_CONFIG_INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PCI_MSIX_TABLE_CONFIG_INTERFACE** structure enables device drivers to modify their MSI-X interrupt settings. This structure describes the GUID_MSIX_TABLE_CONFIG_INTERFACE interface.

Syntax

```
typedef struct _PCI_MSIX_TABLE_CONFIG_INTERFACE {
  USHORT                      Size;
  USHORT                      Version;
  PVOID                       Context;
  PINTERFACE_REFERENCE        InterfaceReference;
  PINTERFACE_DEREFERENCE      InterfaceDereference;
  PPCI_MSIX_SET_ENTRY         SetTableEntry;
  PPCI_MSIX_MASKUNMASK_ENTRY MaskTableEntry;
  PPCI_MSIX_MASKUNMASK_ENTRY UnmaskTableEntry;
  PPCI_MSIX_GET_ENTRY         GetTableEntry;
  PPCI_MSIX_GET_TABLE_SIZE    GetTableSize;
} PCI_MSIX_TABLE_CONFIG_INTERFACE, *PPCI_MSIX_TABLE_CONFIG_INTERFACE;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined interface version.

`Context`

A pointer to interface-specific context information.

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the interface's reference count.

`SetTableEntry`

A pointer to the interface's SetTableEntry routine.

`MaskTableEntry`

A pointer to the interface's MaskTableEntry routine.

`UnmaskTableEntry`

A pointer to the interface's UnmaskTableEntry routine.

`GetTableEntry`

Reserved for future use.

`GetTableSize`

Reserved for future use.

Remarks

A driver obtains a pointer to the **PCI_MSIX_TABLE_CONFIG_INTERFACE** structure by sending an IRP_MN_QUERY_INTERFACE IRP to its bus driver with **InterfaceType** set to GUID_MSIX_TABLE_CONFIG_INTERFACE.

Requirements

Minimum supported client	Available in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

GUID_MSIX_TABLE_CONFIG_INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

MaskTableEntry

SetTableEntry

UnmaskTableEntry

PCI_SLOT_NUMBER structure

1/30/2019 • 2 minutes to read • Edit Online

The **PCI_SLOT_NUMBER** structure is obsolete. It defines the format of the *Slot* parameter to the obsolete **Hal*Xxx* BusData** routines when they are called with the *BusDataType* value **PCIConfiguration**.

Syntax

```
typedef struct _PCI_SLOT_NUMBER {
  union {
    struct {
      ULONG DeviceNumber : 5;
      ULONG FunctionNumber : 3;
      ULONG Reserved : 24;
    } bits;
    ULONG AsULONG;
  } u;
} PCI_SLOT_NUMBER, *PPCI_SLOT_NUMBER;
```

Members

`u`

`u.bits`

`u.bits.DeviceNumber`

`u.bits.FunctionNumber`

`u.bits.Reserved`

`u.AsULONG`

Specifies the logical slot number of the device being configured.

Remarks

Drivers of PCI devices can call HalGetBusData or HalGetBusDataByOffset more than once for the same slot number to get the configuration information for their device(s).

For example, a driver might search for devices it supports on all PCI buses in the machine first, and then call `HalGetBusData(ByOffset)` again to request more configuration information about devices of interest. Such a driver could code a loop that calls `HalGetBusData(ByOffset)` with an input `Buffer` of sufficient `Length` only to contain enough of the PCI_COMMON_CONFIG to determine the `VendorID` and `DeviceID` of each PCI device. After finding the `Slot` numbers for any promising PCI devices, the driver would call HalGetBusData or HalGetBusDataByOffset one or more times with additional buffer space to get the information needed to configure its devices.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)

See Also

HalAssignSlotResources

HalGetBusData

HalGetBusDataByOffset

HalSetBusData

HalSetBusDataByOffset

PCI_COMMON_CONFIG

PLUGPLAY_NOTIFICATION_HEADER structure

1/30/2019 • 2 minutes to read • Edit Online

A **PLUGPLAY_NOTIFICATION_HEADER** structure is included at the beginning of each PnP notification structure, such as a DEVICE_INTERFACE_CHANGE_NOTIFICATION structure.

Syntax

```
typedef struct _PLUGPLAY_NOTIFICATION_HEADER {
  USHORT Version;
  USHORT Size;
  GUID   Event;
} PLUGPLAY_NOTIFICATION_HEADER, *PPLUGPLAY_NOTIFICATION_HEADER;
```

Members

`Version`

Specifies the version of the data structure, currently set to 1.

`Size`

Specifies the size of the structure, in bytes.

`Event`

Specifies a GUID identifying the event.

Remarks

Drivers can cast a PnP notification structure to this type to access the **Event** field and identify the exact type of the structure.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION

HWPROFILE_CHANGE_NOTIFICATION

IoRegisterPlugPlayNotification

TARGET_DEVICE_CUSTOM_NOTIFICATION

TARGET_DEVICE_REMOVAL_NOTIFICATION

PNP_BUS_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **PNP_BUS_INFORMATION** structure describes a bus.

Syntax

```
typedef struct _PNP_BUS_INFORMATION {
  GUID           BusTypeGuid;
  INTERFACE_TYPE LegacyBusType;
  ULONG          BusNumber;
} PNP_BUS_INFORMATION, *PPNP_BUS_INFORMATION;
```

Members

`BusTypeGuid`

`LegacyBusType`

`BusNumber`

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IRP_MN_QUERY_BUS_INFORMATION

PO_FX_COMPONENT_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_IDLE_STATE** structure specifies the attributes of an Fx power state of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_IDLE_STATE {
  ULONGLONG TransitionLatency;
  ULONGLONG ResidencyRequirement;
  ULONG     NominalPower;
} PO_FX_COMPONENT_IDLE_STATE, *PPO_FX_COMPONENT_IDLE_STATE;
```

Members

`TransitionLatency`

The transition latency. This latency is the amount of time, in 100-nanosecond units, that the component requires to return from this Fx state to the F0 state. For a **PO_FX_COMPONENT_IDLE_STATE** structure that specifies the attributes of the F0 state, set this member to zero.

`ResidencyRequirement`

The residency requirement. The residency requirement is the minimum amount of time, in 100-nanosecond units, that the component must spend in this Fx state to make a transition to this Fx state worthwhile. PoFx uses this member value as a hint to avoid switching a component to an Fx state unless the component is likely to remain in this state for at least the amount of time specified by **ResidencyRequirement**. For a **PO_FX_COMPONENT_IDLE_STATE** structure that describes the attributes of the F0 state, set this member to zero.

`NominalPower`

The power, in microwatts, that the component consumes in this Fx state. Set this member to **PO_FX_UNKNOWN_POWER** to indicate that PoFx should ignore (treat as negligible) the component's internal power consumption in this Fx state when PoFx evaluates which power state to switch to when the component is idle.

Remarks

The PO_FX_COMPONENT structure contains a pointer to an array of **PO_FX_COMPONENT_IDLE_STATE** structures. Each array element specifies the attributes of an Fx state. Element 0 describes F0, element 1 describes F1, and so on.

When the driver registers a device with PoFx, the driver supplies an array of **PO_FX_COMPONENT** structures. Each array element describes the power attributes of a component in the device.

Do not set **TransitionLatency** and **ResidencyRequirement** to **PO_FX_UNKNOWN_TIME** because it may prevent the power management framework (PoFx) from selecting the idle state.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT

PO_FX_COMPONENT_PERF_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_PERF_INFO** structure describes all the sets of performance states for a single component within a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_PERF_INFO {
  ULONG                    PerfStateSetsCount;
  PO_FX_COMPONENT_PERF_SET PerfStateSets[ANYSIZE_ARRAY];
} PO_FX_COMPONENT_PERF_INFO, *PPO_FX_COMPONENT_PERF_INFO;
```

Members

`PerfStateSetsCount`

The number of elements in the array **PerfStateSets** array.

`PerfStateSets`

This member is the first element in an array of one or more PO_FX_COMPONENT_PERF_SET elements. Each **PO_FX_COMPONENT_PERF_SET** represents one set of supported performance states for a component. If the array contains more than one element, the additional elements immediately follow the first element.

Remarks

When a device driver calls the PoFxRegisterComponentPerfStates routine to register a component for performance state support with the power management framework (PoFx), the driver can supply a **PO_FX_COMPONENT_PERF_INFO** structure that describes the sets of the performance states supported by the component. The **PerfStateSets** member of the **PO_FX_COMPONENT_PERF_INFO** structure contains an array of PO_FX_COMPONENT_PERF_SET structures that represent the supported performance states.

Most drivers are expected to define a single set of performance states per component. For example, a driver might define one set of performance states to control the clock frequency for a component. However, some drivers may need to define more than one performance state set to control multiple dimensions of performance states for a component. For example, a driver might define two sets of performance states to control the clock frequency and bus bandwidth.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

Component-Level Performance State Management

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

PO_FX_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_PERF_SET** structure represents a set of performance states for a single component within a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_PERF_SET {
  UNICODE_STRING        Name;
  ULONGLONG             Flags;
  PO_FX_PERF_STATE_UNIT Unit;
  PO_FX_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG             Count;
      PPO_FX_PERF_STATE States;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PO_FX_COMPONENT_PERF_SET, *PPO_FX_COMPONENT_PERF_SET;
```

Members

`Name`

An optional character string used to describe the component property controlled by the performance state set. For example, "Clock frequency" or "Memory bandwidth".

`Flags`

Set to 0. Currently, no flags are defined for this member.

`Unit`

A PO_FX_PERF_STATE_UNIT value that specifies the type of unit that is controlled by the performance state set.

`Type`

A PO_FX_PERF_STATE_TYPE value that specifies the type of performance states in the set (a discrete number of states or a continuous distribution of states between a minimum and maximum value).

`Discrete`

`Discrete.Count`

The number of performance states in the set.

`Discrete.States`

A pointer to a PO_FX_PERF_STATE array. The length of this array is specified by the **Count** member. Each array element describes one power state in the set.

`Range`

`Range.Minimum`

The minimum performance state value.

`Range.Maximum`

The maximum performance state value.

Remarks

When a device driver calls the PoFxRegisterComponentPerfStates routine to register a component for performance state support with the power management framework (PoFx), the driver can supply a PO_FX_COMPONENT_PERF_INFO structure that describes the sets of the performance states supported by the component. The **PerfStateSets** member of the **PO_FX_COMPONENT_PERF_INFO** structure contains an array of **PO_FX_COMPONENT_PERF_SET** structures that represent the supported performance states.

Most drivers are expected to define a single set of performance states per component. For example, a driver might define one set of performance states to control the clock frequency for a component. However, some drivers may need to define more than one performance state set to control multiple dimensions of performance states for a component. For example, a driver might define two sets of performance states to control the clock frequency and bus bandwidth.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PO_FX_COMPONENT_PERF_INFO

PoFxRegisterComponentPerfStates

PO_FX_COMPONENT_V1 structure

1/30/2019 • 3 minutes to read • Edit Online

The **PO_FX_COMPONENT** structure describes the power state attributes of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_V1 {
  GUID                          Id;
  ULONG                         IdleStateCount;
  ULONG                         DeepestWakeableIdleState;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
} PO_FX_COMPONENT_V1, *PPO_FX_COMPONENT_V1;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The driver should specify a nonzero value for this member if the power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of Fx power states that the component supports. A component must support at least one Fx state (F0).

`DeepestWakeableIdleState`

The index of the deepest Fx state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStates`

A pointer to a PO_FX_COMPONENT_IDLE_STATE array. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an Fx power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

Remarks

When a device driver registers a device with PoFx, the driver supplies a PO_FX_DEVICE structure that holds the registration information. This structure contains an array of **PO_FX_COMPONENT** structures. The elements in this array describe the power attributes of the individual components in the device. PoFx uses the information in this array to manage the power settings of these components.

The **Id** member contains a component ID that PoFx can use to uniquely identify a component. Do not confuse the component ID with the component index, which a routine such as PoFxActivateComponent uses to identify a component in a registered device. A component index is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes are integer values in the range 0 to N–1. In contrast, a component ID is a

GUID value.

The **Flags** member may be set to the **PO_FX_COMPONENT_FLAG_F0_ON_DX** value if the driver specified support for **PO_FX_VERSION_V2** during registration with PoFx (that is, during the call to the PoFxRegisterDevice routine, the driver specified the value **PO_FX_VERSION_V2** for the **Version** member of the PO_FX_DEVICE structure that was passed to the *Device* parameter). If the **PO_FX_COMPONENT_FLAG_F0_ON_DX** flag is specified, PoFx will place the component into the F0 power state during Dx transitions and for wait/wake IRPs. Before dispatching a Dx IRP or wait/wake IRP to the device stack (whichever event happens first), the power manager will transition the component into the F0 state. The component will be kept in F0 until the wake/wait IRP is canceled or completed, or the Dx IRP is completed and driver has called the PoFxReportDevicePoweredOn routine (whichever event happens last).

Component dependencies

PoFx supports logical component dependencies within a device (but not across different devices). During device registration for each component, the driver can supply a number (the *ProviderCount* member) and an array of component indexes (the *Providers* member) this component depends on; these are known as *providers*. PoFx guarantees that the component will not become active until all of its providers are active.

Component dependencies form a directed graph where an edge points from a dependent component to a provider and represents a dependency. During device registration, PoFx validates the dependency graph: cycles and repeated edges are not allowed and the depth of the graph for the depth-first search traversal cannot exceed four. PoFx also sets the components' active reference count according to the dependency graph; that is, each provider's active reference count is incremented once for each component that depends on it.

During the component activation, PoFx asynchronously activates all providers first. After a component is active, PoFx checks every dependent component and continues their activation if necessary.

When the component transitions to the idle state, PoFx first makes the dependent components idle. After the driver completes the ComponentIdleConditionCallback and the platform extension plug-in (PEP) is notified that the component is idle, PoFx recursively traverses the dependency graph in breadth-first traversal and asynchronously idles each provider.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wudfwdm.h)

See Also

Component-Level Power Management

PO_FX_COMPONENT_IDLE_STATE

PO_FX_DEVICE

PO_FX_COMPONENT_V2 structure

1/30/2019 • 4 minutes to read • Edit Online

The **PO_FX_COMPONENT** structure describes the power state attributes of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_V2 {
  GUID                          Id;
  ULONGLONG                     Flags;
  ULONG                         DeepestWakeableIdleState;
  ULONG                         IdleStateCount;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
  ULONG                         ProviderCount;
  PULONG                        Providers;
} PO_FX_COMPONENT_V2, *PPO_FX_COMPONENT_V2;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The driver should specify a nonzero value for this member if the power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`Flags`

This member can be set to the following flag bit: **PO_FX_COMPONENT_FLAG_F0_ON_DX**. For more information, see Remarks.

`DeepestWakeableIdleState`

The index of the deepest Fx state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of Fx power states that the component supports. A component must support at least one Fx state (F0).

`IdleStates`

A pointer to a PO_FX_COMPONENT_IDLE_STATE array. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an Fx power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

`ProviderCount`

A value that specifies the number of components this component depends on. For more information, see *Component dependencies* under Remarks.

`Providers`

A pointer to an array of component indexes this component depends on. For more information, see *Component dependencies* under Remarks.

Remarks

When a device driver registers a device with PoFx, the driver supplies a PO_FX_DEVICE structure that holds the registration information. This structure contains an array of **PO_FX_COMPONENT** structures. The elements in this array describe the power attributes of the individual components in the device. PoFx uses the information in this array to manage the power settings of these components.

The **Id** member contains a component ID that PoFx can use to uniquely identify a component. Do not confuse the component ID with the component index, which a routine such as PoFxActivateComponent uses to identify a component in a registered device. A component index is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes are integer values in the range 0 to N–1. In contrast, a component ID is a GUID value.

The **Flags** member may be set to the **PO_FX_COMPONENT_FLAG_F0_ON_DX** value if the driver specified support for **PO_FX_VERSION_V2** during registration with PoFx (that is, during the call to the PoFxRegisterDevice routine, the driver specified the value **PO_FX_VERSION_V2** for the **Version** member of the PO_FX_DEVICE structure that was passed to the *Device* parameter). If the **PO_FX_COMPONENT_FLAG_F0_ON_DX** flag is specified, PoFx will place the component into the F0 power state during Dx transitions and for wait/wake IRPs. Before dispatching a Dx IRP or wait/wake IRP to the device stack (whichever event happens first), the power manager will transition the component into the F0 state. The component will be kept in F0 until the wake/wait IRP is canceled or completed, or the Dx IRP is completed and driver has called the PoFxReportDevicePoweredOn routine (whichever event happens last).

Component dependencies

PoFx supports logical component dependencies within a device (but not across different devices). During device registration for each component, the driver can supply a number (the *ProviderCount* member) and an array of component indexes (the *Providers* member) this component depends on; these are known as *providers*. PoFx guarantees that the component will not become active until all of its providers are active.

Component dependencies form a directed graph where an edge points from a dependent component to a provider and represents a dependency. During device registration, PoFx validates the dependency graph: cycles and repeated edges are not allowed and the depth of the graph for the depth-first search traversal cannot exceed four. PoFx also sets the components' active reference count according to the dependency graph; that is, each provider's active reference count is incremented once for each component that depends on it.

During the component activation, PoFx asynchronously activates all providers first. After a component is active, PoFx checks every dependent component and continues their activation if necessary.

When the component transitions to the idle state, PoFx first makes the dependent components idle. After the driver completes the ComponentIdleConditionCallback and the platform extension plug-in (PEP) is notified that the component is idle, PoFx recursively traverses the dependency graph in breadth-first traversal and asynchronously idles each provider.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wudfwdm.h)

See Also

Component-Level Power Management

PO_FX_COMPONENT_IDLE_STATE

PO_FX_DEVICE

PO_FX_DEVICE_V1 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_DEVICE** structure describes the power attributes of a device to the power management framework (PoFx).

Syntax

```
typedef struct _PO_FX_DEVICE_V1 {
  ULONG                                    Version;
  ULONG                                    ComponentCount;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK   ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK       ComponentIdleStateCallback;
  PPO_FX_DEVICE_POWER_REQUIRED_CALLBACK      DevicePowerRequiredCallback;
  PPO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK  DevicePowerNotRequiredCallback;
  PPO_FX_POWER_CONTROL_CALLBACK              PowerControlCallback;
  PVOID                                    DeviceContext;
  PO_FX_COMPONENT_V1                       Components[ANYSIZE_ARRAY];
} PO_FX_DEVICE_V1, *PPO_FX_DEVICE_V1;
```

Members

`Version`

The version number of this structure. If the driver will register for performance state support with PoFx, set this member to **PO_FX_VERSION_V2**. Otherwise, set this member to **PO_FX_VERSION_V1**.

`ComponentCount`

The number of elements in the **Components** array. Additionally, this member specifies the number of components in the device.

`ComponentActiveConditionCallback`

A pointer to the ComponentActiveConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleConditionCallback`

A pointer to the ComponentIdleConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleStateCallback`

A pointer to the ComponentIdleStateCallback callback routine that is implemented by the device driver.

`DevicePowerRequiredCallback`

A pointer to the DevicePowerRequiredCallback callback routine that is implemented by the device driver.

`DevicePowerNotRequiredCallback`

A pointer to the DevicePowerNotRequiredCallback callback routine that is implemented by the device driver.

`PowerControlCallback`

A pointer to the PowerControlCallback callback routine that is implemented by the device driver.

`DeviceContext`

A pointer to a caller-allocated device context. This pointer is passed as a parameter to each driver-implemented callback function that is pointed to by this structure. The device driver uses this context to store information about the current power state of the device. This context is opaque to PoFx.

`Components`

This member is the first element in an array of one or more PO_FX_COMPONENT elements. If the array contains more than one element, the additional elements immediately follow the **PO_FX_DEVICE** structure. The array contains one element for each component in the device. The Fx power state of each component can be controlled independently of the Fx power states of the other components in the device. The **ANYSIZE_ARRAY** constant is defined to be 1 in the Ntdef.h header file.

Remarks

To register a device with PoFx, a driver calls the PoFxRegisterDevice routine and supplies, as a parameter, a pointer to a **PO_FX_DEVICE** structure that describes the device.

Each element in the **Components** array describes the power state attributes of one component in the device. Each component in the device is identified by its **Components** array index. Routines such as PoFxActivateComponent and PoFxCompleteIdleCondition use the array index of a component to identify the component.

A device driver is not required to implement all six callback routines. The driver can set a function pointer in the **PO_FX_DEVICE** structure to NULL if the driver does not implement the corresponding callback routine. However, certain callback routines must be implemented. Specifically, if one or more components in a device has more than one Fx state, the driver must implement ComponentIdleStateCallback, ComponentActiveConditionCallback, and ComponentIdleConditionCallback routines. Otherwise, device registration fails and PoFxRegisterDevice returns **STATUS_INVALID_PARAMETER**.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wudfwdm.h)

See Also

ComponentActiveConditionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

DevicePowerNotRequiredCallback

DevicePowerRequiredCallback

PO_FX_COMPONENT

PoFxActivateComponent

PoFxCompleteIdleCondition

PoFxRegisterDevice

PowerControlCallback

PO_FX_DEVICE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_DEVICE** structure describes the power attributes of a device to the power management framework (PoFx).

Syntax

```
typedef struct _PO_FX_DEVICE_V2 {
  ULONG                                    Version;
  ULONGLONG                                Flags;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK   ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK       ComponentIdleStateCallback;
  PPO_FX_DEVICE_POWER_REQUIRED_CALLBACK      DevicePowerRequiredCallback;
  PPO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK  DevicePowerNotRequiredCallback;
  PPO_FX_POWER_CONTROL_CALLBACK              PowerControlCallback;
  PVOID                                    DeviceContext;
  ULONG                                    ComponentCount;
  PO_FX_COMPONENT_V2                       Components[ANYSIZE_ARRAY];
} PO_FX_DEVICE_V2, *PPO_FX_DEVICE_V2;
```

Members

`Version`

The version number of this structure. If the driver will register for performance state support with PoFx, set this member to **PO_FX_VERSION_V2**. Otherwise, set this member to **PO_FX_VERSION_V1**.

`Flags`

Set to 0. Currently, no flags are defined for this member.

`ComponentActiveConditionCallback`

A pointer to the ComponentActiveConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleConditionCallback`

A pointer to the ComponentIdleConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleStateCallback`

A pointer to the ComponentIdleStateCallback callback routine that is implemented by the device driver.

`DevicePowerRequiredCallback`

A pointer to the DevicePowerRequiredCallback callback routine that is implemented by the device driver.

`DevicePowerNotRequiredCallback`

A pointer to the DevicePowerNotRequiredCallback callback routine that is implemented by the device driver.

`PowerControlCallback`

A pointer to the PowerControlCallback callback routine that is implemented by the device driver.

`DeviceContext`

A pointer to a caller-allocated device context. This pointer is passed as a parameter to each driver-implemented callback function that is pointed to by this structure. The device driver uses this context to store information about the current power state of the device. This context is opaque to PoFx.

`ComponentCount`

The number of elements in the **Components** array. Additionally, this member specifies the number of components in the device.

`Components`

This member is the first element in an array of one or more PO_FX_COMPONENT elements. If the array contains more than one element, the additional elements immediately follow the **PO_FX_DEVICE** structure. The array contains one element for each component in the device. The Fx power state of each component can be controlled independently of the Fx power states of the other components in the device. The **ANYSIZE_ARRAY** constant is defined to be 1 in the Ntdef.h header file.

Remarks

To register a device with PoFx, a driver calls the PoFxRegisterDevice routine and supplies, as a parameter, a pointer to a **PO_FX_DEVICE** structure that describes the device.

Each element in the **Components** array describes the power state attributes of one component in the device. Each component in the device is identified by its **Components** array index. Routines such as PoFxActivateComponent and PoFxCompleteIdleCondition use the array index of a component to identify the component.

A device driver is not required to implement all six callback routines. The driver can set a function pointer in the **PO_FX_DEVICE** structure to NULL if the driver does not implement the corresponding callback routine. However, certain callback routines must be implemented. Specifically, if one or more components in a device has more than one Fx state, the driver must implement ComponentIdleStateCallback, ComponentActiveConditionCallback, and ComponentIdleConditionCallback routines. Otherwise, device registration fails and PoFxRegisterDevice returns **STATUS_INVALID_PARAMETER**.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wdm.h (include Wudfwdm.h)

See Also

ComponentActiveConditionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

DevicePowerNotRequiredCallback

DevicePowerRequiredCallback

PO_FX_COMPONENT

PoFxActivateComponent

PoFxCompleteIdleCondition

PoFxRegisterDevice

PowerControlCallback

PO_FX_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE** structure represents a performance state for a single component within a device.

Syntax

```
typedef struct _PO_FX_PERF_STATE {
  ULONGLONG Value;
  PVOID     Context;
} PO_FX_PERF_STATE, *PPO_FX_PERF_STATE;
```

Members

`Value`

The value of this performance state. The units are specified by the **Unit** member of the PO_FX_COMPONENT_PERF_SET that contains this performance state. For example, if **Value** is 100000000 and the **Unit** member of the **PO_FX_COMPONENT_PERF_SET** is **PoFxPerfStateUnitFrequency**, this performance state represents 100 MHz.

`Context`

A pointer to additional context for the performance state that cannot be presented by the **Value** member. This data can be shared between the driver and the platform extension plug-in (PEP).

> **Note** Using the **Context** member can hinder the visualization of performance states in Windows Performance Analyzer.

Remarks

The **States** member of the PO_FX_COMPONENT_PERF_SET structure contains an array of **PO_FX_PERF_STATE** elements.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PO_FX_COMPONENT_PERF_SET

PO_FX_PERF_STATE_UNIT

PoFxRegisterComponentPerfStates

PO_FX_PERF_STATE_CHANGE structure

2/5/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_CHANGE** structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.

Syntax

```
typedef struct _PO_FX_PERF_STATE_CHANGE {
  ULONG Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PO_FX_PERF_STATE_CHANGE, *PPO_FX_PERF_STATE_CHANGE;
```

Members

`Set`

The index of the performance state set that is being changed within the collection of performance state sets for the component.

`StateIndex`

For sets that represent a discrete number of performance states (that is, where the **Type** member of the PO_FX_COMPONENT_PERF_SET is **PoFxPerfStateTypeDiscrete**), this member is the index of the performance state to use for the new performance level.

`StateValue`

For sets that represent a continuous distribution of performance states (that is, where the **Type** member of the PO_FX_COMPONENT_PERF_SET is **PoFxPerfStateTypeRange**), this member is the value of the performance state to use for the new performance level.

Remarks

The **PO_FX_PERF_STATE_CHANGE** structure is used for the *PerfChange* parameter of the PoFxIssueComponentPerfStateChange routine and the *PerfChanges* parameter of the PoFxIssueComponentPerfStateChangeMultiple routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PoFxIssueComponentPerfStateChange

PoFxIssueComponentPerfStateChangeMultiple

PO_FX_PERF_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_TYPE** enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.

Syntax

```
typedef enum _PO_FX_PERF_STATE_TYPE {
  PoFxPerfStateTypeDiscrete,
  PoFxPerfStateTypeRange,
  PoFxPerfStateTypeMaximum
} PO_FX_PERF_STATE_TYPE, *PPO_FX_PERF_STATE_TYPE;
```

Constants

PoFxPerfStateTypeDiscrete	Indicates that the performance state set contains a discrete number of states.
PoFxPerfStateTypeRange	Indicates that the performance state set contains a continuous distribution of states between a minimum and maximum value.
PoFxPerfStateTypeMaximum	This value is reserved for system use.

Remarks

The **Type** member of the PO_FX_COMPONENT_PERF_SET structure is a value from the **PO_FX_PERF_STATE_TYPE** enumeration.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

PO_FX_PERF_STATE_UNIT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_UNIT** enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.

Syntax

```
typedef enum _PO_FX_PERF_STATE_UNIT {
  PoFxPerfStateUnitOther,
  PoFxPerfStateUnitFrequency,
  PoFxPerfStateUnitBandwidth,
  PoFxPerfStateUnitMaximum
} PO_FX_PERF_STATE_UNIT, *PPO_FX_PERF_STATE_UNIT;
```

Constants

PoFxPerfStateUnitOther	Indicates that the performance states control a unit other than frequency or bandwidth.
PoFxPerfStateUnitFrequency	Indicates that the performance states control frequency, measured in Hz.
PoFxPerfStateUnitBandwidth	Indicates that the performance states control bandwidth, measured in bits per second.
PoFxPerfStateUnitMaximum	This value is reserved for system use.

Remarks

The **Unit** member of the PO_FX_COMPONENT_PERF_SET structure is a value from the **PO_FX_PERF_STATE_UNIT** enumeration.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

POOL_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POOL_TYPE** enumeration type specifies the type of system memory to allocate.

Syntax

```
typedef enum _POOL_TYPE {
  NonPagedPool,
  NonPagedPoolExecute,
  PagedPool,
  NonPagedPoolMustSucceed,
  DontUseThisType,
  NonPagedPoolCacheAligned,
  PagedPoolCacheAligned,
  NonPagedPoolCacheAlignedMustS,
  MaxPoolType,
  NonPagedPoolBase,
  NonPagedPoolBaseMustSucceed,
  NonPagedPoolBaseCacheAligned,
  NonPagedPoolBaseCacheAlignedMustS,
  NonPagedPoolSession,
  PagedPoolSession,
  NonPagedPoolMustSucceedSession,
  DontUseThisTypeSession,
  NonPagedPoolCacheAlignedSession,
  PagedPoolCacheAlignedSession,
  NonPagedPoolCacheAlignedMustSSession,
  NonPagedPoolNx,
  NonPagedPoolNxCacheAligned,
  NonPagedPoolSessionNx
} POOL_TYPE;
```

Constants

NonPagedPool	*Nonpaged pool*, which is nonpageable system memory. Nonpaged pool can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary. System memory allocated with the **NonPagedPool** pool type is executable. For more information, see the description of the **NonPagedPoolExecute** pool type. Starting with Windows 8, drivers should allocate most or all of their nonpaged memory from the no-execute (NX) nonpaged pool instead of the executable nonpaged pool. For more information, see the description of the **NonPagedPoolNx** pool type.

NonPagedPoolExecute	Starting with Windows 8, **NonPagedPoolExecute** is an alternate name for the **NonPagedPool** value. This value indicates that the allocated memory is to be nonpaged and executable—that is, instruction execution is enabled in this memory. To port a driver from an earlier version of Windows, you should typically replace all or most instances of the **NonPagedPool** name in the driver source code with **NonPagedPoolNx**. Avoid replacing instances of the **NonPagedPool** name with **NonPagedPoolExecute** except in cases in which executable memory is explicitly required. For more information, see No-Execute (NX) Nonpaged Pool.
PagedPool	*Paged pool*, which is pageable system memory. Paged pool can only be allocated and accessed at IRQL < DISPATCH_LEVEL.
NonPagedPoolMustSucceed	This value is for internal use only, and is allowed only during system startup. Drivers must not specify this value at times other than system startup, because a "must succeed" request crashes the system if the requested memory size is unavailable.
DontUseThisType	Reserved for system use.
NonPagedPoolCacheAligned	Nonpaged pool, aligned on processor cache boundaries. This value is for internal use only.
PagedPoolCacheAligned	Paged pool, aligned on processor cache boundaries. This value is for internal use only.
NonPagedPoolCacheAlignedMustS	This value is for internal use only, and is allowed only during system startup. It is the cache-aligned equivalent of **NonPagedPoolMustSucceed**.
MaxPoolType	Reserved for system use.
NonPagedPoolBase	Reserved for system use.
NonPagedPoolBaseMustSucceed	Reserved for system use.
NonPagedPoolBaseCacheAligned	Reserved for system use.
NonPagedPoolBaseCacheAlignedMustS	Reserved for system use.
NonPagedPoolSession	Deprecated. Do not use.
PagedPoolSession	Deprecated. Do not use.
NonPagedPoolMustSucceedSession	Deprecated. Do not use.
DontUseThisTypeSession	Deprecated. Do not use.
NonPagedPoolCacheAlignedSession	Deprecated. Do not use.
PagedPoolCacheAlignedSession	Deprecated. Do not use.

NonPagedPoolCacheAlignedMustSSession	Deprecated. Do not use.
NonPagedPoolNx	*No-execute* (NX) nonpaged pool. This pool type is available starting with Windows 8. In contrast to the nonpaged pool designated by **NonPagedPool**, which allocates executable memory, the NX nonpaged pool allocates memory in which instruction execution is disabled. For more information, see No-Execute (NX) Nonpaged Pool. Nonpaged pool can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary.
NonPagedPoolNxCacheAligned	NX nonpaged pool, aligned on processor cache boundaries. This value is reserved for exclusive use by the operating system.
NonPagedPoolSessionNx	Reserved for exclusive use by the operating system.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExAllocatePoolWithTagPriority

ExInitializeLookasideListEx

ExInitializeNPagedLookasideList

ExInitializePagedLookasideList

LookasideListAllocateEx

SeAssignSecurity

SeAssignSecurityEx

POWER_PLATFORM_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_PLATFORM_INFORMATION** structure contains information about the power capabilities of the system.

Syntax

```
typedef struct _POWER_PLATFORM_INFORMATION {
  BOOLEAN AoAc;
} POWER_PLATFORM_INFORMATION, *PPOWER_PLATFORM_INFORMATION;
```

Members

`AoAc`

Indicates whether the system supports the connected standby power model.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Header	wdm.h (include Ntpoapi.h)

POWER_REQUEST_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_REQUEST_TYPE** enumeration indicates the power request type.

Syntax

```
typedef enum _POWER_REQUEST_TYPE {
  PowerRequestDisplayRequired,
  PowerRequestSystemRequired,
  PowerRequestAwayModeRequired,
  PowerRequestExecutionRequired
} POWER_REQUEST_TYPE, *PPOWER_REQUEST_TYPE;
```

Constants

PowerRequestDisplayRequired	Not used by drivers. For more information, see Remarks.
PowerRequestSystemRequired	Prevents the computer from automatically entering sleep mode after a period of user inactivity.
PowerRequestAwayModeRequired	Not used by drivers. For more information, see Remarks.
PowerRequestExecutionRequired	Not used by drivers. For more information, see Remarks.

Remarks

This enumeration is used by the kernel-mode PoClearPowerRequest and PoSetPowerRequest routines. Drivers that call these routines must specify the **PowerRequestSystemRequired** enumeration value.

The other three enumeration values—**PowerRequestDisplayRequired**, **PowerRequestAwayModeRequired**, and **PowerRequestExecutionRequired**—are not used by drivers. Applications specify these power request types in calls to the PowerSetRequest and PowerClearRequest functions.

A **PowerRequestDisplayRequired** power request has the following effects:

- After a period of user inactivity, the session display stays on and will not automatically turn off. If the display is already turned off, the power request turns the display on.
- A screensaver will not automatically start after a period of user inactivity. If a screensaver is already running, the power request stops the screensaver.
- The session will not be automatically locked after a period of user inactivity. If the session is already locked when the driver sends the power request, the session remains locked.

While a **PowerRequestAwayModeRequired** power request is in effect, if the user tries to put the computer into sleep mode (for example, by clicking **Start** and then clicking **Sleep**), the power manager turns off audio and video so that the computer appears to be in sleep mode, but the computer continues to run.

While a **PowerRequestExecutionRequired** power request is in effect, the calling process continues to run instead of being suspended or terminated by process lifetime management (PLM) mechanisms. When and how long the

process is allowed to run depends on the operating system and power policy settings. This type of power request is supported starting with Windows 8.

Requirements

Minimum supported client	Supported starting with Windows 7.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

PoClearPowerRequest

PoSetPowerRequest

PowerClearRequest

PowerSetRequest

POWER_STATE union

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_STATE** union specifies a system power state value or a device power state value.

Syntax

```
typedef union _POWER_STATE {
  SYSTEM_POWER_STATE SystemState;
  DEVICE_POWER_STATE DeviceState;
} POWER_STATE, *PPOWER_STATE;
```

Members

`SystemState`

A system power state value of type SYSTEM_POWER_STATE.

`DeviceState`

A device power state value of type DEVICE_POWER_STATE.

Remarks

The **POWER_STATE** union is used in conjunction with the POWER_STATE_TYPE enumeration type to specify a system power state value or a device power state value. This union is also used in cases where the power state type is implicitly determined by the context in which it is used. For example, see PoRequestPowerIrp and PoSetPowerState.

For more information about power management, see Introduction to Power Management.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

POWER_STATE_TYPE

PoRequestPowerIrp

PoSetPowerState

SYSTEM_POWER_STATE

POWER_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POWER_STATE_TYPE** enumeration type indicates that a power state value is a system power state or a device power state.

Syntax

```
typedef enum _POWER_STATE_TYPE {
  SystemPowerState,
  DevicePowerState
} POWER_STATE_TYPE, *PPOWER_STATE_TYPE;
```

Constants

SystemPowerState	Indicates a SYSTEM_POWER_STATE value.
DevicePowerState	Indicates a DEVICE_POWER_STATE value.

Remarks

The **POWER_STATE_TYPE** enumeration type is used in conjunction with a value of type POWER_STATE to indicate that the power state value is a system power state value or a device power state value. For an example, see PoSetPowerState.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

POWER_STATE

PoSetPowerState

SYSTEM_POWER_STATE

PRIVILEGE_SET structure

1/30/2019 • 2 minutes to read • Edit Online

The **PRIVILEGE_SET** structure specifies a set of security privileges. For more information, see the reference page for **PRIVILEGE_SET** in the Microsoft Windows SDK documentation.

Syntax

```
typedef struct _PRIVILEGE_SET {
  ULONG               PrivilegeCount;
  ULONG               Control;
  LUID_AND_ATTRIBUTES Privilege[ANYSIZE_ARRAY];
} PRIVILEGE_SET, *PPRIVILEGE_SET;
```

Members

`PrivilegeCount`

`Control`

`Privilege`

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

SeAccessCheck

REENUMERATE_SELF_INTERFACE_STANDARD structure

1/30/2019 • 2 minutes to read • Edit Online

The **REENUMERATE_SELF_INTERFACE_STANDARD** interface structure enables a driver to request that its parent bus driver reenumerate the driver's device. This structure defines the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface.

Syntax

```
typedef struct _REENUMERATE_SELF_INTERFACE_STANDARD {
  USHORT                 Size;
  USHORT                 Version;
  PVOID                  Context;
  PINTERFACE_REFERENCE   InterfaceReference;
  PINTERFACE_DEREFERENCE InterfaceDereference;
  PREENUMERATE_SELF      SurpriseRemoveAndReenumerateSelf;
} REENUMERATE_SELF_INTERFACE_STANDARD, *PREENUMERATE_SELF_INTERFACE_STANDARD;
```

Members

`Size`

The size, in bytes, of this structure.

`Version`

The driver-defined interface version.

`Context`

A pointer to interface-specific context information.

`InterfaceReference`

A pointer to an InterfaceReference routine that increments the interface's reference count.

`InterfaceDereference`

A pointer to an InterfaceDereference routine that decrements the interface's reference count.

`SurpriseRemoveAndReenumerateSelf`

A pointer to a ReenumerateSelf routine that requests device reenumeration.

Remarks

A driver obtains a pointer to the **REENUMERATE_SELF_INTERFACE_STANDARD** structure by sending an IRP_MN_QUERY_INTERFACE IRP to its bus driver with **InterfaceType** set to GUID_REENUMERATE_SELF_INTERFACE_STANDARD.

The **REENUMERATE_SELF_INTERFACE_STANDARD** structure is an extension of the INTERFACE structure.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h)

See Also

GUID_REENUMERATE_SELF_INTERFACE_STANDARD

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

InterfaceReference

ReenumerateSelf

REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION** structure contains information that a driver's RegistryCallback routine can use to free resources that the driver previously allocated for the context that is associated with a registry object.

Syntax

```
typedef struct _REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION {
  PVOID Object;
  PVOID ObjectContext;
  PVOID Reserved;
} REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION, *PREG_CALLBACK_CONTEXT_CLEANUP_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key to be deleted.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext.

`Reserved`

This member is reserved for future use.

Remarks

This structure is used by the RegistryCallback routine. When the *Argument1* parameter to the *RegistryCallback* routine is **RegNtCallbackObjectContextCleanup**, the *Argument2* parameter is a pointer to a **REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION** structure.

The driver's *RegistryCallback* routine will receive a **RegNtCallbackObjectContextCleanup** notification after the key object's handle has been closed. In response to this notification, the routine should release any resources that it allocated for the object's context. The **RegNtCallbackObjectContextCleanup** notification occurs when the driver has called CmUnRegisterCallback or the driver's *RegistryCallback* routine has just finished processing a **RegNtPreKeyHandleClose** notification.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

CmUnRegisterCallback

RegistryCallback

REG_CREATE_KEY_INFORMATION structure

1/30/2019 • 3 minutes to read • Edit Online

The **REG_CREATE_KEY_INFORMATION** structure contains information that a driver's RegistryCallback routine can use when a registry key that is being created.

> **Note** Starting with Windows 7, the V1 version of this structure, REG_CREATE_KEY_INFORMATION_V1, is used instead.

Syntax

```
typedef struct _REG_CREATE_KEY_INFORMATION {
  PUNICODE_STRING CompleteName;
  PVOID           RootObject;
  PVOID           ObjectType;
  ULONG           CreateOptions;
  PUNICODE_STRING Class;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
  ACCESS_MASK     DesiredAccess;
  ACCESS_MASK     GrantedAccess;
  PULONG          Disposition;
  PVOID           *ResultObject;
  PVOID           CallContext;
  PVOID           RootObjectContext;
  PVOID           Transaction;
  PVOID           Reserved;
} REG_CREATE_KEY_INFORMATION, REG_OPEN_KEY_INFORMATION, *PREG_CREATE_KEY_INFORMATION,
*PREG_OPEN_KEY_INFORMATION;
```

Members

`CompleteName`

A pointer to a UNICODE_STRING structure that contains the path of the new registry key. The path can be absolute or relative. If the path is absolute, this structure contains a fully qualified path that starts with the "\" character. For an absolute path, the **RootObject** member specifies the **\REGISTRY** key, which is the root directory of the registry tree. If the path is relative, the path starts with a character other than "\", and is relative to the key that is specified by the *RootObject* member.

`RootObject`

Pointer to the registry key object that serves as the root for the path specified by the *CompleteName* member.

`ObjectType`

The **ObjectType** member is reserved for internal use. Drivers must not access this member. This member is defined starting with Windows Vista.

`CreateOptions`

A bitwise OR of flags. For more information about these flags, see the *CreateOptions* parameter of the ZwCreateKey routine. This member is defined starting with Windows Vista.

`Class`

A pointer to a UNICODE_STRING structure that identifies the key's object class. For more information about this member, see the *Class* parameter of the ZwCreateKey routine. This member is defined starting with Windows Vista.

`SecurityDescriptor`

A pointer to a SECURITY_DESCRIPTOR structure that contains security information for the key object. This member is defined starting with Windows Vista.

`SecurityQualityOfService`

A pointer to a **SECURITY_QUALITY_OF_SERVICE** structure, which is defined in Winnt.h. This member is defined starting with Windows Vista.

`DesiredAccess`

The access mask that was specified by the thread that is attempting to create the registry key. This member is defined starting with Windows Vista.

`GrantedAccess`

An access mask that indicates the access rights that have been granted to the thread that is attempting to create the registry key. For more information about this member, see Remarks. This member is defined starting with Windows Vista.

`Disposition`

A value that indicates whether the registry key was created. For more information about this member, see the *Disposition* parameter of the ZwCreateKey routine and the following Remarks section. This member is defined starting with Windows Vista.

`ResultObject`

A pointer to a location that receives the address of the key object that represents the created registry key. For more information about this member, see Remarks. This member is defined starting with Windows Vista.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined starting with Windows Vista.

`RootObjectContext`

A pointer to a driver-defined context information that the driver has associated with the root of the path for the registry object by calling CmSetCallbackObjectContext. This member is defined starting with Windows Vista.

`Transaction`

A pointer to a transaction object that the operation is attempted on. If this member is **NULL**, the operation is being performed in non-transactional context. This member is defined starting with Windows Vista.

`Reserved`

Starting with Windows Vista, if this member is 1, then it is safe to cast this structure to type REG_CREATE_KEY_INFORMATION_V1, which contains additional parameters.

In versions of Windows before Windows Vista, this member is always 0.

Remarks

The configuration manager passes this structure to the RegistryCallback routine every time a thread attempts to create a key—for example, when a user-mode thread calls **RegCreateKey** or **RegCreateKeyEx** or when a driver

calls ZwCreateKey.

If the driver's *RegistryCallback* routine returns STATUS_CALLBACK_BYPASS for a **RegNtPreCreateKeyEx** notification, the driver must supply the **GrantedAccess**, **Disposition**, and **ResultObject** values.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Server 2003, but some structure members are available only in Windows Vista and later versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

REG_CREATE_KEY_INFORMATION_V1

REG_POST_OPERATION_INFORMATION

RegistryCallback

SECURITY_DESCRIPTOR

UNICODE_STRING

ZwCreateKey

REG_CREATE_KEY_INFORMATION_V1 structure

1/30/2019 • 5 minutes to read • Edit Online

The **REG_CREATE_KEY_INFORMATION_V1** structure contains information that a filter driver's RegistryCallback routine can use when a registry key is being created.

Syntax

```
typedef struct _REG_CREATE_KEY_INFORMATION_V1 {
  PUNICODE_STRING CompleteName;
  PVOID           RootObject;
  PVOID           ObjectType;
  ULONG           Options;
  PUNICODE_STRING Class;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
  ACCESS_MASK     DesiredAccess;
  ACCESS_MASK     GrantedAccess;
  PULONG          Disposition;
  PVOID           *ResultObject;
  PVOID           CallContext;
  PVOID           RootObjectContext;
  PVOID           Transaction;
  ULONG_PTR       Version;
  PUNICODE_STRING RemainingName;
  ULONG           Wow64Flags;
  ULONG           Attributes;
  KPROCESSOR_MODE CheckAccessMode;
} REG_CREATE_KEY_INFORMATION_V1, REG_OPEN_KEY_INFORMATION_V1, *PREG_CREATE_KEY_INFORMATION_V1,
*PREG_OPEN_KEY_INFORMATION_V1;
```

Members

`CompleteName`

A pointer to a UNICODE_STRING structure that contains the path of the new registry key. The path can be absolute or relative. If the path is absolute, this structure contains a fully qualified path that starts with the "\" character. For an absolute path, the **RootObject** member specifies the **\REGISTRY** key, which is the root directory of the registry tree. If the path is relative, the path starts with a character other than "\", and is relative to the key that is specified by the **RootObject** member.

`RootObject`

A pointer to a registry key object that represents the root registry key for the path that is specified by the **CompleteName** member.

`ObjectType`

This member is reserved for use by the operating system. Drivers must not access this member.

`Options`

Specifies the options for the key-create routine to use to create or open the new key. For more information, see the description of the *CreateOptions* parameter of the ZwCreateKey routine and the description of the *OpenOptions* parameter of the ZwOpenKeyEx routine.

`Class`

A pointer to a **UNICODE_STRING** structure that identifies the object class of the new key. For more information about this member, see the *Class* parameter of the **ZwCreateKey** routine. This pointer value can be **NULL**.

`SecurityDescriptor`

A pointer to a SECURITY_DESCRIPTOR structure that contains security information for the key object. This pointer was obtained from the **SecurityDescriptor** member of the OBJECT_ATTRIBUTES structure that was passed as an input parameter in the call to create the new registry key.

`SecurityQualityOfService`

A pointer to a SECURITY_QUALITY_OF_SERVICE structure. This structure indicates whether a server can impersonate the client that is trying to create the registry key, and, if impersonation is permitted, the extent to which it is permitted.

`DesiredAccess`

The access mask that was specified by the thread that is trying to create the registry key. For more information about this access mask, see the description of the *DesiredAccess* parameter of the **ZwCreateKey** routine.

`GrantedAccess`

An access mask that indicates the access rights that were granted to the thread that is trying to create the registry key. For more information about this member, see the following Remarks section.

`Disposition`

A value that indicates whether the requested registry operation will create a new key or open an existing one. For more information about this member, see the description of the *Disposition* parameter of the **ZwCreateKey** routine and the following Remarks section.

`ResultObject`

A pointer to a location that receives the address of the key object that represents the created registry key.

`CallContext`

Optional driver-defined context information that the driver's *RegistryCallback* routine can supply.

`RootObjectContext`

A pointer to driver-defined context information that the driver has associated with the root of the path of the registry object by calling the CmSetCallbackObjectContext routine.

`Transaction`

A pointer to a transaction object for the registry operation. You can supply this pointer to the ObOpenObjectByPointer routine to obtain the corresponding transaction handle. If this member is **NULL**, the operation is being performed in non-transactional context.

`Version`

The structure version number. This member distinguishes the REG_CREATE_KEY_INFORMATION structure in Windows Vista from the **REG_CREATE_KEY_INFORMATION_V1** structure in Windows 7 and later versions of Windows. The following version numbers are currently defined.

VERSION NUMBER	VERSION OF STRUCTURE
0	**REG_CREATE_KEY_INFORMATION**

1	**REG_CREATE_KEY_INFORMATION_V1**

Future versions of this structure might add new members but will not change the members that are already defined in existing versions of the structure. This member is defined in the **REG_CREATE_KEY_INFORMATION_V1** structure that is supported in Windows 7 and later versions of the Windows operating systems. In the **REG_CREATE_KEY_INFORMATION** structure that Windows Vista supports, this member is named **Reserved** and is set to zero. Filter drivers should rely on the version number and not the operating system version to determine which version of the structure they are using.

`RemainingName`

A pointer to a **UNICODE_STRING** structure that contains the relative path of the new registry key. This member always expresses the path of the new key relative to the path of the key that is specified by the **RootObject** member. In contrast, the **CompleteName** member can contain an absolute path if the **RootObject** member specifies the **\REGISTRY** key.

`Wow64Flags`

Contains the Wow64 flags from the access mask that was passed as an input parameter in the call to create the new registry key. This member indicates whether a 32-bit client program that is running on a 64-bit version of Windows is trying to create a registry key. This member is set to zero or to one of the following flag bits:

- KEY_WOW64_32KEY
- KEY_WOW64_64KEY

These flag bits are defined in the Wdm.h and Winnt.h header files. For more information about these flags, see Registry Key Security and Access Rights.

`Attributes`

Contains the object-attribute flags from the **Attributes** member of the **OBJECT_ATTRIBUTES** structure that was passed as an input parameter in the call to create the new registry key. This member might contain one or more of the following flag bits:

- OBJ_KERNEL_HANDLE
- OBJ_FORCE_ACCESS_CHECK
- OBJ_OPENLINK

For more information about these flags, see OBJECT_ATTRIBUTES.

`CheckAccessMode`

Indicates how the configuration manager performs the security access check for the call to create the new key. This member contains one of the following MODE enumeration values from the Wdm.h header file:

- **KernelMode**
- **UserMode**

This security check is similar to that performed by the SeAccessCheck routine, which has an *AccessMode* parameter that can be set to either **UserMode** or **KernelMode**. If **CheckAccessMode** is set to **UserMode**, the configuration manager performs a full security access check regardless of whether the call originated in user mode or kernel mode. For more information about how to force user-mode security access checks on a call that originates in kernel mode, see the description of the OBJ_FORCE_ACCESS_CHECK flag in the **Attributes** member of the **OBJECT_ATTRIBUTES** structure.

Remarks

The configuration manager passes this structure to the *RegistryCallback* routine every time that a thread tries to create a key—for example, when a user-mode thread calls RegCreateKey or RegCreateKeyEx, or when a kernel-mode driver calls ZwCreateKey.

This structure is an extended version of the REG_CREATE_KEY_INFORMATION structure that Windows Vista supports. The first 14 members, **CompleteName** through **Transaction**, are identical in the two structures. The last five members of the **REG_CREATE_KEY_INFORMATION_V1** structure, **Version** through **CheckAccessMode**, are not part of the **REG_CREATE_KEY_INFORMATION** structure.

If the driver's *RegistryCallback* routine returns STATUS_CALLBACK_BYPASS for a **RegNtPreCreateKeyEx** notification, the driver must supply the values for the **GrantedAccess**, **Disposition**, and **ResultObject** members.

The **REG_CREATE_KEY_INFORMATION_V1** structure is one of a number of structures that a filter driver can receive through its *RegistryCallback* routine. For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Windows 7 and later versions of the Windows operating systems.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

OBJECT_ATTRIBUTES

ObOpenObjectByPointer

REG_CREATE_KEY_INFORMATION

RegCreateKey

RegCreateKeyEx

RegistryCallback

SECURITY_DESCRIPTOR

SECURITY_QUALITY_OF_SERVICE

UNICODE_STRING

ZwCreateKey

REG_DELETE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_DELETE_KEY_INFORMATION** structure contains information that a driver's RegistryCallback routine can use when a registry key is being deleted.

Syntax

```
typedef struct _REG_DELETE_KEY_INFORMATION {
  PVOID Object;
  PVOID CallContext;
  PVOID ObjectContext;
  PVOID Reserved;
} REG_DELETE_KEY_INFORMATION, *PREG_DELETE_KEY_INFORMATION, REG_FLUSH_KEY_INFORMATION,
*PREG_FLUSH_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key to be deleted.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to delete a key—for example, when a user-mode thread calls **RegDeleteKey** or **RegDeleteKeyEx** or when a driver calls ZwDeleteKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available for Microsoft Windows XP and later versions of the Windows operating system, but some structure members are available only for Windows Vista and later versions.

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

ZwDeleteKey

REG_DELETE_VALUE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_DELETE_VALUE_KEY_INFORMATION** structure contains information that a driver's RegistryCallback routine can use when a registry key's value is being deleted.

Syntax

```
typedef struct _REG_DELETE_VALUE_KEY_INFORMATION {
  PVOID           Object;
  PUNICODE_STRING ValueName;
  PVOID           CallContext;
  PVOID           ObjectContext;
  PVOID           Reserved;
} REG_DELETE_VALUE_KEY_INFORMATION, *PREG_DELETE_VALUE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose value entry is about to be deleted.

`ValueName`

A pointer to a UNICODE_STRING structure that contains the name of the value entry that is about to be deleted.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to delete a value entry—for example, when a user-mode thread calls **RegDeleteValue** or when a driver calls ZwDeleteValueKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

UNICODE_STRING

ZwDeleteValueKey

REG_ENUMERATE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_ENUMERATE_KEY_INFORMATION** structure describes one subkey of a key whose subkeys are being enumerated.

Syntax

```
typedef struct _REG_ENUMERATE_KEY_INFORMATION {
  PVOID                 Object;
  ULONG                 Index;
  KEY_INFORMATION_CLASS KeyInformationClass;
  PVOID                 KeyInformation;
  ULONG                 Length;
  PULONG                ResultLength;
  PVOID                 CallContext;
  PVOID                 ObjectContext;
  PVOID                 Reserved;
} REG_ENUMERATE_KEY_INFORMATION, *PREG_ENUMERATE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose subkeys are being enumerated.

`Index`

The zero-based index of the subkey within the key.

`KeyInformationClass`

The KEY_INFORMATION_CLASS value that indicates the type of information to be returned by the system in the **KeyInformation** buffer.

`KeyInformation`

A pointer to a buffer that contains the information to be returned by the system. The format of the buffer depends on the value of **KeyInformationClass**. For more information, see KEY_INFORMATION_CLASS.

`Length`

The size, in bytes, of the **KeyInformation** buffer.

`ResultLength`

A pointer to a ULONG that receives (from the system) the amount of valid data, in bytes, in the **KeyInformation** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to a RegistryCallback routine every time a thread attempts to enumerate the subkeys of a key—for example, when a user-mode thread calls **RegEnumKey** or **RegEnumKeyEx** or when a driver calls ZwEnumerateKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

KEY_INFORMATION_CLASS

RegistryCallback

ZwEnumerateKey

REG_ENUMERATE_VALUE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_ENUMERATE_VALUE_KEY_INFORMATION** structure describes one value entry of a key whose value entries are being enumerated.

Syntax

```
typedef struct _REG_ENUMERATE_VALUE_KEY_INFORMATION {
  PVOID                       Object;
  ULONG                       Index;
  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass;
  PVOID                       KeyValueInformation;
  ULONG                       Length;
  PULONG                      ResultLength;
  PVOID                       CallContext;
  PVOID                       ObjectContext;
  PVOID                       Reserved;
} REG_ENUMERATE_VALUE_KEY_INFORMATION, *PREG_ENUMERATE_VALUE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose value entries are being enumerated.

`Index`

The zero-based index of the subkey within the key.

`KeyValueInformationClass`

The KEY_VALUE_INFORMATION_CLASS value that indicates the type of information to be returned by the system in the **KeyValueInformation** buffer.

`KeyValueInformation`

A pointer to a buffer that contains the information to be returned by the system. The format of the buffer depends on the value of **KeyValueInformationClass**. For more information see KEY_VALUE_INFORMATION_CLASS.

`Length`

The size, in bytes, of the **KeyValueInformation** buffer.

`ResultLength`

A pointer to a variable that receives (from the system) the amount of valid data, in bytes, in the **KeyValueInformation** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to a RegistryCallback routine every time a thread attempts to enumerate a value entry of a key—for example, when a user-mode thread calls **RegEnumValue** or when a driver calls ZwEnumerateValueKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

KEY_VALUE_INFORMATION_CLASS

RegistryCallback

ZwEnumerateValueKey

REG_KEY_HANDLE_CLOSE_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_KEY_HANDLE_CLOSE_INFORMATION** structure contains information about a registry key whose handle is about to be closed.

Syntax

```
typedef struct _REG_KEY_HANDLE_CLOSE_INFORMATION {
  PVOID Object;
  PVOID CallContext;
  PVOID ObjectContext;
  PVOID Reserved;
} REG_KEY_HANDLE_CLOSE_INFORMATION, *PREG_KEY_HANDLE_CLOSE_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the handle that is about to be closed.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

ZwClose

REG_LOAD_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_LOAD_KEY_INFORMATION** structure contains information about a registry hive that is being loaded.

Syntax

```
typedef struct _REG_LOAD_KEY_INFORMATION {
  PVOID           Object;
  PUNICODE_STRING KeyName;
  PUNICODE_STRING SourceFile;
  ULONG           Flags;
  PVOID           TrustClassObject;
  PVOID           UserEvent;
  ACCESS_MASK     DesiredAccess;
  PHANDLE         RootHandle;
  PVOID           CallContext;
  PVOID           ObjectContext;
  PVOID           Reserved;
} REG_LOAD_KEY_INFORMATION, *PREG_LOAD_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the root key of the hive that is about to be loaded.

`KeyName`

A pointer to a UNICODE_STRING structure that contains the name of the root key.

`SourceFile`

A pointer to a **UNICODE_STRING** structure that contains the path name of a file that contains the registry hive information that is being loaded.

`Flags`

Reserved for system use.

`TrustClassObject`

Reserved for system use.

`UserEvent`

A pointer to an event object that is signaled when the hive is unloaded.

`DesiredAccess`

The access mask that was specified by the thread that is trying to load the registry key. For more information about this access mask, see the description of the *DesiredAccess* parameter of the ZwCreateKey routine.

`RootHandle`

An optional pointer to a location that receives the handle to the root of the hive that is being loaded. This member can be non-**NULL** if an application hive is being loaded. In all other cases, this member should be **NULL**.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling the CmSetCallbackObjectContext routine.

`Reserved`

Reserved for system use.

Remarks

The operating system passes the **REG_LOAD_KEY_INFORMATION** structure to a RegistryCallback routine every time a user-mode thread calls RegLoadKey to load a registry hive.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegLoadKey

RegistryCallback

UNICODE_STRING

ZwCreateKey

REG_NOTIFY_CLASS Enumeration

1/30/2019 • 11 minutes to read • Edit Online

The **REG_NOTIFY_CLASS** enumeration type specifies the type of registry operation that the configuration manager is passing to a RegistryCallback routine.

Syntax

```
typedef enum _REG_NOTIFY_CLASS {
  RegNtDeleteKey,
  RegNtPreDeleteKey,
  RegNtSetValueKey,
  RegNtPreSetValueKey,
  RegNtDeleteValueKey,
  RegNtPreDeleteValueKey,
  RegNtSetInformationKey,
  RegNtPreSetInformationKey,
  RegNtRenameKey,
  RegNtPreRenameKey,
  RegNtEnumerateKey,
  RegNtPreEnumerateKey,
  RegNtEnumerateValueKey,
  RegNtPreEnumerateValueKey,
  RegNtQueryKey,
  RegNtPreQueryKey,
  RegNtQueryValueKey,
  RegNtPreQueryValueKey,
  RegNtQueryMultipleValueKey,
  RegNtPreQueryMultipleValueKey,
  RegNtPreCreateKey,
  RegNtPostCreateKey,
  RegNtPreOpenKey,
  RegNtPostOpenKey,
  RegNtKeyHandleClose,
  RegNtPreKeyHandleClose,
  RegNtPostDeleteKey,
  RegNtPostSetValueKey,
  RegNtPostDeleteValueKey,
  RegNtPostSetInformationKey,
  RegNtPostRenameKey,
  RegNtPostEnumerateKey,
  RegNtPostEnumerateValueKey,
  RegNtPostQueryKey,
  RegNtPostQueryValueKey,
  RegNtPostQueryMultipleValueKey,
  RegNtPostKeyHandleClose,
  RegNtPreCreateKeyEx,
  RegNtPostCreateKeyEx,
  RegNtPreOpenKeyEx,
  RegNtPostOpenKeyEx,
  RegNtPreFlushKey,
  RegNtPostFlushKey,
  RegNtPreLoadKey,
  RegNtPostLoadKey,
  RegNtPreUnLoadKey,
  RegNtPostUnLoadKey,
  RegNtPreQueryKeySecurity,
  RegNtPostQueryKeySecurity,
  RegNtPreSetKeySecurity,
  RegNtPostSetKeySecurity,
  RegNtCallbackObjectContextCleanup,
  RegNtPreRestoreKey,
  RegNtPostRestoreKey,
  RegNtPreSaveKey,
  RegNtPostSaveKey,
  RegNtPreReplaceKey,
  RegNtPostReplaceKey,
  RegNtPreQueryKeyName,
  RegNtPostQueryKeyName,
  MaxRegNtNotifyClass
} REG_NOTIFY_CLASS;
```

Constants

RegNtDeleteKey	Specifies that a thread is attempting to delete a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreDeleteKey	Specifies that a thread is attempting to delete a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtSetValueKey	Specifies that a thread is attempting to set a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreSetValueKey	Specifies that a thread is attempting to set a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtDeleteValueKey	Specifies that a thread is attempting to delete a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreDeleteValueKey	Specifies that a thread is attempting to delete a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtSetInformationKey	Specifies that a thread is attempting to set the metadata for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreSetInformationKey	Specifies that a thread is attempting to set the metadata for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtRenameKey	Specifies that a thread is attempting to rename a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreRenameKey	Specifies that a thread is attempting to rename a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtEnumerateKey	Specifies that a thread is attempting to enumerate a subkey of a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreEnumerateKey	Specifies that a thread is attempting to enumerate a subkey of a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtEnumerateValueKey	Specifies that a thread is attempting to enumerate a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.

RegNtPreEnumerateValueKey	Specifies that a thread is attempting to enumerate a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtQueryKey	Specifies that a thread is attempting to read the metadata for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreQueryKey	Specifies that a thread is attempting to read the metadata for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtQueryValueKey	Specifies that a thread is attempting to read a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.
RegNtPreQueryValueKey	Specifies that a thread is attempting to read a value entry for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtQueryMultipleValueKey	Specifies that a thread is attempting to query multiple value entries for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on only Windows XP.
RegNtPreQueryMultipleValueKey	Specifies that a thread is attempting to query multiple value entries for a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPreCreateKey	Specifies that a thread is attempting to create a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows XP and later versions of the Windows operating system.
RegNtPostCreateKey	Specifies that a thread has successfully created a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows XP and later versions of the Windows operating system.
RegNtPreOpenKey	Specifies that a thread is attempting to open an existing key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows XP and later versions of the Windows operating system.
RegNtPostOpenKey	Specifies that a thread has successfully opened an existing key. This value indicates a post-notification call to *RegistryCallback* . Use this value on Windows XP and later versions of the Windows operating system.
RegNtKeyHandleClose	Specifies that a thread is attempting to close a key handle. This value indicates a pre-notification call to *RegistryCallback*. Use this value only on Windows XP.

RegNtPreKeyHandleClose	Specifies that a thread is attempting to close a key handle. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system. Drivers can monitor this registry operation but they cannot block or modify it.
RegNtPostDeleteKey	Specifies that the system has attempted to delete the key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostSetValueKey	Specifies that the system has attempted to set a value entry for a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostDeleteValueKey	Specifies that the system has attempted to delete a value entry for a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostSetInformationKey	Specifies that the system has attempted to set the key's metadata. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostRenameKey	Specifies that the system has attempted to rename the key. This value indicates a post-notification call to *RegistryCallback* . Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostEnumerateKey	Specifies that the system has attempted to enumerate the subkey of a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostEnumerateValueKey	Specifies that the system has attempted to enumerate the value entry of a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostQueryKey	Specifies that the system has attempted to query the metadata for a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostQueryValueKey	Specifies that the system has attempted to query a value entry for the key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostQueryMultipleValueKey	Specifies that the system has attempted to query multiple value entries for the key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.

RegNtPostKeyHandleClose	Specifies that the system has attempted to close a key handle. This value indicates a post-notification call to *RegistryCallback* . Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPreCreateKeyEx	Specifies that a thread is attempting to create a key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostCreateKeyEx	Specifies that the system has attempted to create a key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPreOpenKeyEx	Specifies that a thread is attempting to open an existing key. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPostOpenKeyEx	Specifies that the system has attempted to open an existing key. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Server 2003 and later versions of the Windows operating system.
RegNtPreFlushKey	Specifies that a thread is attempting to write a key to disk. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPostFlushKey	Specifies that the system has attempted to write a key to disk. This value indicates a post-notification call to *RegistryCallback* . Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPreLoadKey	Specifies that a thread is attempting to load a registry hive from a file. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPostLoadKey	Specifies that the system has attempted to load a registry hive from a file. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPreUnLoadKey	Specifies that a thread is attempting to unload a registry hive. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPostUnLoadKey	Specifies that the system has attempted to unload a registry hive. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPreQueryKeySecurity	Specifies that a thread is attempting to obtain a registry key's security information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.

RegNtPostQueryKeySecurity	Specifies that a thread has attempted to obtain a registry key's security information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPreSetKeySecurity	Specifies that a thread is attempting to set a registry key's security information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPostSetKeySecurity	Specifies that a thread has attempted to set a registry key's security information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtCallbackObjectContextCleanup	Specifies that the driver has called CmUnRegisterCallback or the driver's RegistryCallback routine has just finished processing a **RegNtPreKeyHandleClose** class value. Use this value on Windows Vista and later versions of the Windows operating system.
RegNtPreRestoreKey	Specifies that a thread is attempting to restore a registry key's information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPostRestoreKey	Specifies that a thread has attempted to restore a registry key's information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPreSaveKey	Specifies that a thread is attempting to save a registry key's information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPostSaveKey	Specifies that a thread has attempted to save a registry key's information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPreReplaceKey	Specifies that a thread is attempting to replace a registry key's information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPostReplaceKey	Specifies that a thread has attempted to replace a registry key's information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.
RegNtPreQueryKeyName	Specifies that a thread is attempting to obtain the full path of a registry key. Use this value on Windows 10 and later versions of the Windows operating system.

RegNtPostQueryKeyName	Specifies that a thread has attempted to obtain the full path of a registry key. Use this value on Windows 10 and later versions of the Windows operating system.
MaxRegNtNotifyClass	Specifies the maximum value in this enumeration type.

Remarks

When the configuration manager calls a driver's *RegistryCallback* routine, it passes a **REG_NOTIFY_CLASS** enumeration value to the routine. The configuration manager also passes a notification-specific structure that contains information about the notification. For a list of these structures, see RegistryCallback.

Requirements

Minimum supported client	Available for Windows XP and later versions of the Windows operating system, but some enumeration values are available only for specified later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmUnRegisterCallback

RegistryCallback

REG_POST_CREATE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_POST_CREATE_KEY_INFORMATION** structure contains the result of an attempt to create a registry key.

Syntax

```
typedef struct _REG_POST_CREATE_KEY_INFORMATION {
  PUNICODE_STRING CompleteName;
  PVOID           Object;
  NTSTATUS        Status;
} REG_POST_CREATE_KEY_INFORMATION, REG_POST_OPEN_KEY_INFORMATION, *PREG_POST_CREATE_KEY_INFORMATION,
*PREG_POST_OPEN_KEY_INFORMATION;
```

Members

`CompleteName`

A pointer to a UNICODE_STRING structure that specifies the complete path of the registry key.

`Object`

A pointer to the registry key object that was created.

`Status`

The NTSTATUS value that the system will return for the registry operation.

Remarks

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available only on Microsoft Windows XP.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

RegistryCallback

UNICODE_STRING

ZwCreateKey

REG_POST_OPERATION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_POST_OPERATION_INFORMATION** structure contains information about a completed registry operation that a RegistryCallback routine can use.

Syntax

```
typedef struct _REG_POST_OPERATION_INFORMATION {
  PVOID    Object;
  NTSTATUS Status;
  PVOID    PreInformation;
  NTSTATUS ReturnStatus;
  PVOID    CallContext;
  PVOID    ObjectContext;
  PVOID    Reserved;
} REG_POST_OPERATION_INFORMATION, *PREG_POST_OPERATION_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for which the operation has completed. This member is valid only if the Status member of the structure is set to STATUS_SUCCESS. For more information, see Invalid Key Object Pointers in Registry Notifications.

`Status`

The NTSTATUS-typed value that the system will return for the registry operation.

`PreInformation`

A pointer to the structure that contains preprocessing information for the registry operation that has completed. For example, if the RegistryCallback routine is processing a **RegNtPostQueryValueKey** operation, the **PreInformation** member points to a REG_QUERY_VALUE_KEY_INFORMATION structure. This member is defined for Windows Vista and later versions of the Windows operating system.

`ReturnStatus`

A driver-supplied NTSTATUS-typed value. If the driver's *RegistryCallback* routine returns STATUS_CALLBACK_BYPASS, the operating system uses the **ReturnStatus** member's value as the status that it returns to the thread that initiated the registry operation. (In such cases, the operating system also copies the **ReturnStatus** member's value to the **Status** member.) Otherwise, this member is ignored. This member is defined for Windows Vista and later versions of the Windows operating system.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows

operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

For more information about handling post-notifications, see Handling Notifications.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows Server 2003 and later versions of the Windows operating system, but some structure members are available only for Windows Vista and later versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

REG_QUERY_VALUE_KEY_INFORMATION

RegistryCallback

REG_PRE_CREATE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_PRE_OPEN_KEY_INFORMATION** structure contains the name of a registry key that is about to be opened.

Syntax

```
typedef struct _REG_PRE_CREATE_KEY_INFORMATION {
  PUNICODE_STRING CompleteName;
} REG_PRE_CREATE_KEY_INFORMATION, REG_PRE_OPEN_KEY_INFORMATION, *PREG_PRE_CREATE_KEY_INFORMATION,
*PREG_PRE_OPEN_KEY_INFORMATION;
```

Members

`CompleteName`

A pointer to a UNICODE_STRING structure that contains the complete path of the registry key.

Remarks

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available only on Microsoft Windows XP.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

RegistryCallback

UNICODE_STRING

ZwOpenKey

REG_QUERY_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_QUERY_KEY_INFORMATION** structure describes the metadata that is about to be queried for a key.

Syntax

```
typedef struct _REG_QUERY_KEY_INFORMATION {
  PVOID                Object;
  KEY_INFORMATION_CLASS KeyInformationClass;
  PVOID                KeyInformation;
  ULONG                Length;
  PULONG               ResultLength;
  PVOID                CallContext;
  PVOID                ObjectContext;
  PVOID                Reserved;
} REG_QUERY_KEY_INFORMATION, *PREG_QUERY_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose metadata is about to be queried.

`KeyInformationClass`

The KEY_INFORMATION_CLASS value that indicates the type of information to be returned by the system.

`KeyInformation`

A pointer to a buffer that contains the information to be returned by the system. The format of the buffer depends on the value of **KeyInformationClass**. For more information see KEY_INFORMATION_CLASS.

`Length`

Specifies the size, in bytes, of the **KeyInformation** buffer.

`ResultLength`

Pointer to a variable that receives (from the system) the amount of valid data, in bytes, in the **KeyInformation** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the

Windows operating system.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to query the metadata for a key—for example, when a user-mode thread calls **RegQueryInfoKey** or when a driver calls ZwQueryKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

ZwQueryKey

REG_QUERY_KEY_NAME structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_QUERY_KEY_NAME** structure describes the full registry key name of an object being queried.

Syntax

```
typedef struct _REG_QUERY_KEY_NAME {
  PVOID                    Object;
  POBJECT_NAME_INFORMATION ObjectNameInfo;
  ULONG                    Length;
  PULONG                   ReturnLength;
  PVOID                    CallContext;
  PVOID                    ObjectContext;
  PVOID                    Reserved;
} REG_QUERY_KEY_NAME, *PREG_QUERY_KEY_NAME;
```

Members

`Object`

A pointer to the registry key object for the key whose metadata is about to be queried.

`ObjectNameInfo`

A pointer to an **OBJECT_NAME_INFORMATION** structure (see wdm.h) that contains the full registry key name to be returned by the system, as a Unicode string.

`Length`

Specifies the size, in bytes, of the **ObjectNameInfo** buffer.

`ReturnLength`

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. It contains the key context for the key that is being queried.

`Reserved`

This member is reserved for future use.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to query the full name of the registry key.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows 10 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

REG_NOTIFY_CLASS

RegistryCallback

REG_QUERY_KEY_SECURITY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_QUERY_KEY_SECURITY_INFORMATION** structure receives security information for a registry key object.

Syntax

```
typedef struct _REG_QUERY_KEY_SECURITY_INFORMATION {
  PVOID                 Object;
  PSECURITY_INFORMATION SecurityInformation;
  PSECURITY_DESCRIPTOR  SecurityDescriptor;
  PULONG                Length;
  PVOID                 CallContext;
  PVOID                 ObjectContext;
  PVOID                 Reserved;
} REG_QUERY_KEY_SECURITY_INFORMATION, *PREG_QUERY_KEY_SECURITY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose security information is being retrieved.

`SecurityInformation`

A pointer to a SECURITY_INFORMATION-typed value that identifies the type of security information that is being requested.

`SecurityDescriptor`

A pointer to a SECURITY_DESCRIPTOR structure that receives security information for the key object.

`Length`

The length, in bytes of the **SECURITY_DESCRIPTOR** structure that **SecurityDescriptor** points to.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes the **REG_QUERY_KEY_SECURITY_INFORMATION** structure to the RegistryCallback routine every time a thread attempts to query a key's security information—for example, when a driver calls ZwQuerySecurityObject.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

SECURITY_DESCRIPTOR

SECURITY_INFORMATION

ZwQuerySecurityObject

REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION** structure describes the multiple value entries that are being retrieved for a key.

Syntax

```
typedef struct _REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION {
  PVOID            Object;
  PKEY_VALUE_ENTRY ValueEntries;
  ULONG            EntryCount;
  PVOID            ValueBuffer;
  PULONG           BufferLength;
  PULONG           RequiredBufferLength;
  PVOID            CallContext;
  PVOID            ObjectContext;
  PVOID            Reserved;
} REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION, *PREG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose value entries are being retrieved.

`ValueEntries`

A pointer to an array of KEY_VALUE_ENTRY structures, one for each value entry that is retrieved.

`EntryCount`

The number of entries in the **ValueEntries** array.

`ValueBuffer`

A pointer to a buffer that receives (from the system) the data for all the value entries specified by the **ValueEntries** array.

`BufferLength`

A pointer to a variable that contains the length, in bytes, of the **ValueBuffer** buffer.

`RequiredBufferLength`

A pointer to a variable that receives (from the system) the number of bytes required to hold the data for all the value entries that the **ValueEntries** array specifies. This member can be **NULL**.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to retrieve multiple value entries for a key at once—for example, when a user-mode thread calls **RegQueryMultipleValues**.

Each KEY_VALUE_ENTRY structure in the **ValueEntries** array describes one value entry in the **ValueBuffer** buffer. Specifically, the **DataOffset** member of **KEY_VALUE_ENTRY** contains the offset within **ValueBuffer** where the data for that value entry begins, and the **DataLength** member contains the length, in bytes, of the data for that value entry.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

KEY_VALUE_ENTRY

RegistryCallback

REG_QUERY_VALUE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_QUERY_VALUE_KEY_INFORMATION** structure contains information about a registry key's value entry that is being queried.

Syntax

```
typedef struct _REG_QUERY_VALUE_KEY_INFORMATION {
  PVOID                       Object;
  PUNICODE_STRING             ValueName;
  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass;
  PVOID                       KeyValueInformation;
  ULONG                       Length;
  PULONG                      ResultLength;
  PVOID                       CallContext;
  PVOID                       ObjectContext;
  PVOID                       Reserved;
} REG_QUERY_VALUE_KEY_INFORMATION, *PREG_QUERY_VALUE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose value entry is being queried.

`ValueName`

A pointer to a UNICODE_STRING structure that contains the name of the value entry to retrieve information for.

`KeyValueInformationClass`

The KEY_VALUE_INFORMATION_CLASS value that indicates the type of information to be returned by the system.

`KeyValueInformation`

A pointer to a buffer that contains the information to be returned by the system. The format of the buffer depends on the value of **KeyValueInformationClass**. For more information see KEY_VALUE_INFORMATION_CLASS.

`Length`

The size, in bytes, of the **KeyValueInformation** buffer.

`ResultLength`

A pointer to a variable that receives (from the system) the amount of valid data, in bytes, in the **KeyValueInformation** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to a RegistryCallback routine every time a thread attempts to query a value entry for a key—for example, when a user-mode thread calls **RegQueryValue** or **RegQueryValueEx** or when a driver calls ZwDeleteKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

ZwDeleteKey

ZwQueryValueKey

REG_RENAME_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_RENAME_KEY_INFORMATION** structure contains the new name for a registry key whose name is about to be changed.

Syntax

```
typedef struct _REG_RENAME_KEY_INFORMATION {
  PVOID           Object;
  PUNICODE_STRING NewName;
  PVOID           CallContext;
  PVOID           ObjectContext;
  PVOID           Reserved;
} REG_RENAME_KEY_INFORMATION, *PREG_RENAME_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose name is about to be changed.

`NewName`

A pointer to a UNICODE_STRING structure that contains the new name for the registry key.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

Note that when a key is renamed, only the last component of the path can be changed.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

REG_REPLACE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_REPLACE_KEY_INFORMATION** structure describes the metadata that is about to be replaced for a key.

Syntax

```
typedef struct _REG_REPLACE_KEY_INFORMATION {
  PVOID           Object;
  PUNICODE_STRING OldFileName;
  PUNICODE_STRING NewFileName;
  PVOID           CallContext;
  PVOID           ObjectContext;
  PVOID           Reserved;
} REG_REPLACE_KEY_INFORMATION, *PREG_REPLACE_KEY_INFORMATION;
```

Members

`Object`

A pointer to a registry key object for the key whose metadata is about to be replaced.

`OldFileName`

The name of the file that receives a backup copy of the registry information being replaced.

`NewFileName`

The name of the file with the registry information. This file is typically created by using the **RegSaveKey** function.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information, which the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to the RegistryCallback routine every time a thread attempts to replace the metadata for a key.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Windows Vista SP2 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

REG_RESTORE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_RESTORE_KEY_INFORMATION** structure contains the information for a registry key that is about to be restored.

Syntax

```
typedef struct _REG_RESTORE_KEY_INFORMATION {
  PVOID  Object;
  HANDLE FileHandle;
  ULONG  Flags;
  PVOID  CallContext;
  PVOID  ObjectContext;
  PVOID  Reserved;
} REG_RESTORE_KEY_INFORMATION, *PREG_RESTORE_KEY_INFORMATION;
```

Members

`Object`

A pointer to a registry key object for the key whose name is about to be changed.

`FileHandle`

A handle to the file from which the hive will be restored.

`Flags`

REG_FORCE_RESTORE

0x00000008L

If specified, the restore operation is executed even if open handles exist at or beneath the location in the registry hierarchy to which the *hKey* parameter points.

REG_WHOLE_HIVE_VOLATILE

0x00000001L

If specified, a new, volatile (memory-only) set of registry information, or *hive*, is created. If REG_WHOLE_HIVE_VOLATILE is specified, the key identified by the *hKey* parameter must be either the HKEY_USERS or HKEY_LOCAL_MACHINE value.

REG_REFRESH_HIVE

0x00000002

If set, the location of the subtree that the *hKey* parameter points to is restored to its state immediately following the last flush. The subtree must not be lazy flushed (by calling **RegRestoreKey** with REG_NO_LAZY_FLUSH specified as the value of this parameter); the caller must have the trusted computing base (TCB) privilege; and the handle to which the *hKey* parameter refers must point to the root of the subtree.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is

defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information, which the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

Note that when a key is restored, only the last component of the path can be changed.

The REG_REFRESH_HIVE flag is opaque and a filter should not attempt to change it.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Windows Vista SP2 and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

REG_SAVE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_SAVE_KEY_INFORMATION** structure contains the information for a registry key that is about to be saved.

Syntax

```
typedef struct _REG_SAVE_KEY_INFORMATION {
  PVOID  Object;
  HANDLE FileHandle;
  ULONG  Format;
  PVOID  CallContext;
  PVOID  ObjectContext;
  PVOID  Reserved;
} REG_SAVE_KEY_INFORMATION, *PREG_SAVE_KEY_INFORMATION;
```

Members

`Object`

A pointer to a registry key object for the key whose information is about to be saved.

`FileHandle`

A handle to the file to which the hive information will be saved.

`Format`

The format in which the key or hive is saved. This member can have one of the following values.

VALUE	MEANING
REG_STANDARD_FORMAT	Save in standard format. The standard format is the only format supported by Windows 2000.
REG_LATEST_FORMAT	Save in the latest format. The latest format is supported starting with Windows XP. After the key is saved in this format, it cannot be loaded on an earlier version of Windows.
REG_NO_COMPRESSION	Save with no compression, for faster save operations. The registry key object must represent the root of a hive.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined starting with Windows Vista.

`ObjectContext`

A pointer to driver-defined context information, which the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined starting with Windows Vista.

`Reserved`

This member is reserved for future use. This member is defined starting with Windows Vista.

Remarks

Note that when a key is saved, only the last component of the path can be changed.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista SP2.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

REG_SET_INFORMATION_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_SET_INFORMATION_KEY_INFORMATION** structure describes a new setting for a key's metadata.

Syntax

```
typedef struct _REG_SET_INFORMATION_KEY_INFORMATION {
  PVOID                     Object;
  KEY_SET_INFORMATION_CLASS KeySetInformationClass;
  PVOID                     KeySetInformation;
  ULONG                     KeySetInformationLength;
  PVOID                     CallContext;
  PVOID                     ObjectContext;
  PVOID                     Reserved;
} REG_SET_INFORMATION_KEY_INFORMATION, *PREG_SET_INFORMATION_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose metadata is about to be changed.

`KeySetInformationClass`

The KEY_SET_INFORMATION_CLASS value that indicates the type of information about to be changed.

`KeySetInformation`

A pointer to a buffer that contains the information about to be written. The format of the buffer depends on the value of **KeySetInformationClass**. For more information, see KEY_SET_INFORMATION_CLASS.

`KeySetInformationLength`

The size, in bytes, of the **KeySetInformation** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

REG_SET_KEY_SECURITY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_SET_KEY_SECURITY_INFORMATION** structure specifies security information for a registry key object.

Syntax

```
typedef struct _REG_SET_KEY_SECURITY_INFORMATION {
  PVOID                 Object;
  PSECURITY_INFORMATION SecurityInformation;
  PSECURITY_DESCRIPTOR  SecurityDescriptor;
  PVOID                 CallContext;
  PVOID                 ObjectContext;
  PVOID                 Reserved;
} REG_SET_KEY_SECURITY_INFORMATION, *PREG_SET_KEY_SECURITY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose security information is being set.

`SecurityInformation`

A pointer to a SECURITY_INFORMATION-typed variable that identifies the type of security information that is being set.

`SecurityDescriptor`

A pointer to a SECURITY_DESCRIPTOR structure that contains security information for the key object.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes the **REG_SET_KEY_SECURITY_INFORMATION** structure to the RegistryCallback routine every time a thread attempts to set a key's security information—for example, when a driver calls ZwSetSecurityObject.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

SECURITY_DESCRIPTOR

SECURITY_INFORMATION

ZwSetSecurityObject

REG_SET_VALUE_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_SET_VALUE_INFORMATION** structure describes a new setting for a registry key's value entry.

Syntax

```
typedef struct _REG_SET_VALUE_KEY_INFORMATION {
  PVOID           Object;
  PUNICODE_STRING ValueName;
  ULONG           TitleIndex;
  ULONG           Type;
  PVOID           Data;
  ULONG           DataSize;
  PVOID           CallContext;
  PVOID           ObjectContext;
  PVOID           Reserved;
} REG_SET_VALUE_KEY_INFORMATION, *PREG_SET_VALUE_KEY_INFORMATION;
```

Members

`Object`

A pointer to the registry key object for the key whose value entry is about to be changed.

`ValueName`

A pointer to a UNICODE_STRING structure that contains the name of the value entry about to be changed.

`TitleIndex`

Reserved for system use. Drivers should ignore this member.

`Type`

The type of data that is about to be written. For more information about the possible values for this member, see the *Type* parameter of ZwSetValueKey.

`Data`

A pointer to a buffer that contains the data about to be written. The format of the buffer depends on the value of **Type**. For more information, see ZwSetValueKey.

`DataSize`

The size, in bytes, of the **Data** buffer.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined for Windows Vista and later versions of the Windows operating system.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined for Windows Vista and later versions of the Windows

operating system.

`Reserved`

This member is reserved for future use. This member is defined for Windows Vista and later versions of the Windows operating system.

Remarks

The system passes this structure to a RegistryCallback routine every time a thread attempts to set a value entry for a key—for example, when a user-mode thread calls **RegSetValue** or **RegSetValueEx** or when a driver calls ZwSetValueKey.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available on Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

ZwSetValueKey

REG_UNLOAD_KEY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **REG_UNLOAD_KEY_INFORMATION** structure contains information that a driver's RegistryCallback routine can use when a registry hive is unloaded.

Syntax

```
typedef struct _REG_UNLOAD_KEY_INFORMATION {
  PVOID Object;
  PVOID UserEvent;
  PVOID CallContext;
  PVOID ObjectContext;
  PVOID Reserved;
} REG_UNLOAD_KEY_INFORMATION, *PREG_UNLOAD_KEY_INFORMATION;
```

Members

`Object`

A pointer to a registry key object. This object represents the root key of the hive that is being unloaded.

`UserEvent`

A pointer to an event object.

`CallContext`

Optional driver-defined context information that the driver's RegistryCallback routine can supply. This member is defined starting with Windows Vista.

`ObjectContext`

A pointer to driver-defined context information that the driver has associated with a registry object by calling CmSetCallbackObjectContext. This member is defined starting with Windows Vista.

`Reserved`

This member is reserved for future use. This member is defined starting with Windows Vista.

Remarks

This structure is used by the RegistryCallback routine. When the *Argument1* parameter to the *RegistryCallback* routine is **RegNtPreUnloadKey**, the *Argument2* parameter is a pointer to a **REG_UNLOAD_KEY_INFORMATION** structure.

The operating system passes the **REG_UNLOAD_KEY_INFORMATION** structure to the *RegistryCallback* routine every time a thread attempts to unload a key—for example, when a user-mode thread calls the RegUnloadKey function. In response to a **RegNtPreUnloadKey** notification, the *RegistryCallback* routine should free any resources the driver previously allocated for use with the hive before the hive is unloaded.

For more information about registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CmSetCallbackObjectContext

RegistryCallback

RESOURCEMANAGER_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **RESOURCEMANAGER_BASIC INFORMATION** structure contains information about a resource manager object.

Syntax

```
typedef struct _RESOURCEMANAGER_BASIC_INFORMATION {
  GUID  ResourceManagerId;
  ULONG DescriptionLength;
  WCHAR Description[1];
} RESOURCEMANAGER_BASIC_INFORMATION, *PRESOURCEMANAGER_BASIC_INFORMATION;
```

Members

`ResourceManagerId`

A GUID that KTM assigned to the resource manager.

`DescriptionLength`

The length, in bytes, of the character array that the **Description** member provides.

`Description`

A caller-allocated array of characters that receives a resource manager's description string.

Remarks

The **RESOURCEMANAGER_BASIC INFORMATION** structure is used with the ZwQueryInformationResourceManager routine.

The resource manager's description string, which the **Description** member receives, is the description that the resource manager previously specified when it called ZwCreateResourceManager.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwCreateResourceManager

ZwQueryInformationResourceManager

RESOURCEMANAGER_COMPLETION_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **RESOURCEMANAGER_COMPLETION_INFORMATION** structure is not used.

Syntax

```
typedef struct _RESOURCEMANAGER_COMPLETION_INFORMATION {
  HANDLE    IoCompletionPortHandle;
  ULONG_PTR CompletionKey;
} RESOURCEMANAGER_COMPLETION_INFORMATION, *PRESOURCEMANAGER_COMPLETION_INFORMATION;
```

Members

`IoCompletionPortHandle`

`CompletionKey`

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntifs.h)

RESOURCEMANAGER_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **RESOURCEMANAGER_INFORMATION_CLASS** enumeration identifies the type of information that the ZwQueryInformationResourceManager routine can retrieve for a resource manager object.

Syntax

```
typedef enum _RESOURCEMANAGER_INFORMATION_CLASS {
  ResourceManagerBasicInformation,
  ResourceManagerCompletionInformation
} RESOURCEMANAGER_INFORMATION_CLASS;
```

Constants

ResourceManagerBasicInformation	Information about a resource manager object is stored in a RESOURCEMANAGER_BASIC_INFORMATION structure.
ResourceManagerCompletionInformation	Information about a resource manager object is stored in a RESOURCEMANAGER_COMPLETION_INFORMATION structure.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

RESOURCEMANAGER_BASIC_INFORMATION

RESOURCEMANAGER_COMPLETION_INFORMATION

ZwQueryInformationResourceManager

SCATTER_GATHER_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **SCATTER_GATHER_LIST** structure describes the scatter/gather list for a DMA operation.

Syntax

```
typedef struct _SCATTER_GATHER_LIST {
  ULONG                  NumberOfElements;
  ULONG_PTR              Reserved;
  SCATTER_GATHER_ELEMENT Elements[];
} SCATTER_GATHER_LIST, *PSCATTER_GATHER_LIST;
```

Members

`NumberOfElements`

Specifies the number of elements in the **Elements** array.

`Reserved`

Reserved for future use.

`Elements`

Specifies an array of scatter/gather elements that comprise a scatter/gather list. Each array element is a structure of type **SCATTER_GATHER_ELEMENT**, which is defined as follows:

```
typedef struct _SCATTER_GATHER_ELEMENT {
  PHYSICAL_ADDRESS  Address;
  ULONG  Length;
  ULONG_PTR  Reserved;
} SCATTER_GATHER_ELEMENT, *PSCATTER_GATHER_ELEMENT;
```

The **SCATTER_GATHER_ELEMENT** structure specifies the length, in bytes, of a physically contiguous scatter/gather region and the starting physical address of this region.

Remarks

For a driver that performs scatter/gather DMA, the GetScatterGatherList or BuildScatterGatherList routine creates a scatter/gather list and passes this list to the driver's AdapterListControl routine. This scatter/gather list consists of a **SCATTER_GATHER_LIST** structure that is immediately followed in memory by a **SCATTER_GATHER_ELEMENT** array. The number of elements in this array is specified by the **NumberOfElements** member of the **SCATTER_GATHER_LIST** structure.

For more information about scatter/gather lists, see Using Scatter/Gather DMA.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

AdapterListControl

BuildScatterGatherList

GetScatterGatherList

SDEV_IDENTIFIER_INTERFACE structure

1/30/2019 • 2 minutes to read • Edit Online

This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.

Syntax

```
typedef struct _SDEV_IDENTIFIER_INTERFACE {
  INTERFACE            InterfaceHeader;
  PGET_SDEV_IDENTIFIER GetIdentifier;
} SDEV_IDENTIFIER_INTERFACE, *PSDEV_IDENTIFIER_INTERFACE;
```

Members

`InterfaceHeader`

`GetIdentifier`

Requirements

Header	wdm.h

SECTION_OBJECT_POINTERS structure

1/30/2019 • 2 minutes to read • Edit Online

The **SECTION_OBJECT_POINTERS** structure, allocated by a file system or a redirector driver, is used by the memory manager and cache manager to store file-mapping and cache-related information for a file stream.

Syntax

```
typedef struct _SECTION_OBJECT_POINTERS {
  PVOID DataSectionObject;
  PVOID SharedCacheMap;
  PVOID ImageSectionObject;
} SECTION_OBJECT_POINTERS;
```

Members

`DataSectionObject`

Opaque pointer to a data section object (that is, a **CONTROL_AREA** structure) that is used to track state information for a data file stream. Memory manager sets this member when the operating system first caches the data stream. A **NULL** value indicates that the data stream is currently not in memory; this value, however, can change at any time.

`SharedCacheMap`

Opaque pointer to a cache map object (that is, a **SHARED_CACHE_MAP** structure) that is used to track views that are mapped to the data file stream. Cache manager sets this member when the operating system first caches the stream. A **NULL** value indicates that the data stream is

```
currently
```

not cached; this value, however, can change at any time.

`ImageSectionObject`

Opaque pointer to an image section object (that is, a **CONTROL_AREA** structure) that is used to track state information for an executable file stream. Memory manager sets this member whenever an executable image section is created for the stream. A **NULL** value indicates that the executable image is currently not in memory; this value, however, can change at any time.

Remarks

The **SECTION_OBJECT_POINTERS** structure links a file object to a file stream's section object. That is, through its members, the **SECTION_OBJECT_POINTERS** structure connects a particular file object to virtual memory control structures that keep track of the stream's contents when they are in memory, and allow the operating system to fetch those contents when they are not.

There is a one-to-one relationship between a **SECTION_OBJECT_POINTERS** structure and a file stream. Multiple file objects can be associated with a particular file stream, each representing an open instance of the stream. However, only one **SECTION_OBJECT_POINTERS** structure can be associated with a given stream. If there are multiple file objects for a stream, the **SectionObjectPointer** member for all file objects must point to the

same **SECTION_OBJECT_POINTERS** structure (that is associated with the stream).

For the first file stream open request, the file system or the redirector driver must:

1. Allocate a **SECTION_OBJECT_POINTERS** structure from a nonpaged pool.
2. Initialize all members of the allocated **SECTION_OBJECT_POINTERS** structure to **NULL**.
3. Set the **SectionObjectPointer** member of the associate file object to point to the initialized **SECTION_OBJECT_POINTERS** structure.

For subsequent open requests to the same file stream, the file system or the redirector driver must set the **SectionObjectPointer** member of the associated file object to point to the previously allocated **SECTION_OBJECT_POINTERS** structure for the file stream.

> **Warning** File system filter drivers must treat the members of the **SECTION_OBJECT_POINTERS** structure as opaque because the underlying file system is responsible for the synchronization of the members and, therefore, could change their values at any time.

> **Note** A file is composed of one or more streams, depending on the file system. For more information, see File Streams, Stream Contexts, and Per-Stream Contexts.

Requirements

Minimum supported client	Available in all versions of Windows that Microsoft supports.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

CcInitializeCacheMap

DEVICE_OBJECT

FILE_OBJECT

FOBX

FSRTL_ADVANCED_FCB_HEADER

IoGetDeviceObjectPointer

MmFlushImageSection

ObDereferenceObject

SLIST_ENTRY structure

1/30/2019 • 2 minutes to read • Edit Online

An **SLIST_ENTRY** structure describes an entry in a sequenced singly linked list.

Syntax

```
typedef struct _SLIST_ENTRY {
  struct _SLIST_ENTRY *Next;
} SLIST_ENTRY, *PSLIST_ENTRY;
```

Members

`Next`

Pointer to the next entry in the list, or **NULL** if there is no next entry in the list.

Remarks

A driver can access the **Next** member of a **SLIST_ENTRY**, but must only be updated by the system routines supplied for this purpose.

On 64-bit platforms, **SLIST_ENTRY** structures must be 16-byte aligned. Drivers can use DECLSPEC_ALIGN(MEMORY_ALLOCATION_ALIGNMENT) to ensure the proper alignment of **SLIST_ENTRY**.

For more information about how to use **SLIST_ENTRY** structures to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ExInterlockedFlushSList

ExInterlockedPopEntrySList

ExInterlockedPushEntrySList

SYSTEM_POWER_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **SYSTEM_POWER_STATE** enumeration type is used to indicate a system power state.

Syntax

```
typedef enum _SYSTEM_POWER_STATE {
  PowerSystemUnspecified,
  PowerSystemWorking,
  PowerSystemSleeping1,
  PowerSystemSleeping2,
  PowerSystemSleeping3,
  PowerSystemHibernate,
  PowerSystemShutdown,
  PowerSystemMaximum
} SYSTEM_POWER_STATE, *PSYSTEM_POWER_STATE;
```

Constants

PowerSystemUnspecified	Indicates an unspecified system power state.
PowerSystemWorking	Indicates maximum system power, which corresponds to system working state S0.
PowerSystemSleeping1	Indicates a system sleeping state less than **PowerSystemWorking** and greater than **PowerSystemSleeping2**, which corresponds to system power state S1.
PowerSystemSleeping2	Indicates a system sleeping state less than **PowerSystemSleeping1** and greater than **PowerSystemSleeping3**, which corresponds to system power state S2.
PowerSystemSleeping3	Indicates a system sleeping state less than **PowerSystemSleeping2** and greater than **PowerSystemHibernate**, which corresponds to system power state S3.
PowerSystemHibernate	Indicates the lowest-powered sleeping state, which corresponds to system power state S4.
PowerSystemShutdown	Indicates the system is turned off, which corresponds to system shutdown state S5.
PowerSystemMaximum	The number of system power state values for this enumeration type that represents actual power states. This value is the number of elements in the **DeviceState** member of the DEVICE_CAPABILITIES structure for a device. The other system power state values are less than this value.

Remarks

A power state indicates the level of power consumption—and thus the extent of computing activity—by the system or by a single device. The power manager sets the power state of the system as a whole, where the system power state is indicated by one of the values of the **SYSTEM_POWER_STATE** enumeration type. Device drivers set the power state of their individual devices, where the device power state is indicated by one of the values of the DEVICE_POWER_STATE enumeration type.

For more information about system power states, see Handling System Power State Requests and for more information about device power states, see Managing Power for Individual Devices.

For more information about power management in general, see Introduction to Power Management.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ntpoapi.h)

See Also

DEVICE_POWER_STATE

SYSTEM_POWER_STATE_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **SYSTEM_POWER_STATE_CONTEXT** structure is a partially opaque system structure that contains information about the previous system power states of a computer.

Syntax

```
typedef struct _SYSTEM_POWER_STATE_CONTEXT {
  union {
    struct {
      ULONG Reserved1 : 8;
      ULONG TargetSystemState : 4;
      ULONG EffectiveSystemState : 4;
      ULONG CurrentSystemState : 4;
      ULONG IgnoreHibernationPath : 1;
      ULONG PseudoTransition : 1;
      ULONG KernelSoftReboot : 1;
      ULONG DirectedDripsTransition : 1;
      ULONG Reserved2 : 8;
    } DUMMYSTRUCTNAME;
    ULONG ContextAsUlong;
  } DUMMYUNIONNAME;
} SYSTEM_POWER_STATE_CONTEXT, *PSYSTEM_POWER_STATE_CONTEXT;
```

Members

`DUMMYUNIONNAME`

Unnamed union.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Reserved1`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.TargetSystemState`

The target system power state of the previous system power IRP that the driver received. This member is set to a SYSTEM_POWER_STATE enumeration value. Drivers should treat this member as read-only.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EffectiveSystemState`

The effective previous system power state, as perceived by the user. This member is set to a **SYSTEM_POWER_STATE** enumeration value. Drivers should treat this member as read-only. This member value might not match the **TargetSystemState** member if, for example, the previous system power IRP indicated that the computer was about to enter hibernation, but a hybrid shutdown instead occurred to prepare the computer for a fast startup. For more information, see Remarks.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.CurrentSystemState`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IgnoreHibernationPath`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.PseudoTransition`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.KernelSoftReboot`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DirectedDripsTransition`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Reserved2`

Opaque member. Reserved for system use.

`DUMMYUNIONNAME.ContextAsUlong`

Opaque member. Reserved for system use.

Remarks

Starting with Windows Vista, the I/O stack location in a system power IRP contains a **SYSTEM_POWER_STATE_CONTEXT** structure. The **Power** member of the **IO_STACK_LOCATION** structure contains a **SystemPowerStateContext** member, which is a **SYSTEM_POWER_STATE_CONTEXT** structure. For more information, see IO_STACK_LOCATION.

The size of the **SYSTEM_POWER_STATE_CONTEXT** structure is four bytes. This structure is divided into bit fields, most of which are opaque to drivers and reserved exclusively for use by the operating system. However, two of these bit fields, **TargetSystemState** and **EffectiveSystemState**, can be read by kernel-mode drivers to distinguish a fast startup from a wake-from-hibernation startup. For more information, see Distinguishing Fast Startup from Wake-from-Hibernation.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Ntpoapi.h)

See Also

IO_STACK_LOCATION

SYSTEM_POWER_STATE

TARGET_DEVICE_CUSTOM_NOTIFICATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TARGET_DEVICE_CUSTOM_NOTIFICATION** structure describes a custom device event.

Syntax

```
typedef struct _TARGET_DEVICE_CUSTOM_NOTIFICATION {
  USHORT       Version;
  USHORT       Size;
  GUID         Event;
  PFILE_OBJECT FileObject;
  LONG         NameBufferOffset;
  UCHAR        CustomDataBuffer[1];
} TARGET_DEVICE_CUSTOM_NOTIFICATION, *PTARGET_DEVICE_CUSTOM_NOTIFICATION;
```

Members

`Version`

Specifies the version of the data structure, currently 1.

`Size`

Specifies the size of the structure, in bytes, including the first three standard members plus the event-specific data.

`Event`

Specifies a GUID identifying the event. GUIDs for custom event notification are defined by the components that use this mechanism.

`FileObject`

Pointer to a file object for the device.

`NameBufferOffset`

Specifies the offset, in bytes, from beginning of **CustomDataBuffer** where text begins. A value of -1 indicates that there is no text.

`CustomDataBuffer`

A variable-length buffer, optionally containing binary data at the start of the buffer, followed by an optional text buffer (word-aligned).

Remarks

Kernel-mode components use this structure for custom event notification: to signal a custom event (**IoReportTargetDeviceChange**[**Asynchronous**]) and when handling a custom event (in a notification callback routine).

This structure accommodates both a variable-length binary data buffer and a variable-length Unicode text buffer. The *NameBufferOffset* must indicate where the text buffer begins, so the data can be delivered in the appropriate format (ANSI or Unicode) to user-mode applications that registered for handle-based notification with

RegisterDeviceNotification. See the Microsoft Windows SDK documentation for information about **RegisterDeviceNotification**.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoRegisterPlugPlayNotification

IoReportTargetDeviceChange

IoReportTargetDeviceChangeAsynchronous

TARGET_DEVICE_REMOVAL_NOTIFICATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TARGET_DEVICE_REMOVAL_NOTIFICATION** structure describes a device-removal event. The PnP manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryTargetDeviceChange** events.

Syntax

```
typedef struct _TARGET_DEVICE_REMOVAL_NOTIFICATION {
  USHORT       Version;
  USHORT       Size;
  GUID         Event;
  PFILE_OBJECT FileObject;
} TARGET_DEVICE_REMOVAL_NOTIFICATION, *PTARGET_DEVICE_REMOVAL_NOTIFICATION;
```

Members

`Version`

Specifies the version of the data structure, currently set to 1.

`Size`

Specifies the size of the structure, in bytes, including the size of the standard first three members plus the event-specific data.

`Event`

Specifies a GUID identifying the event: GUID_TARGET_DEVICE_QUERY_REMOVE, GUID_TARGET_DEVICE_REMOVE_COMPLETE, or GUID_TARGET_DEVICE_REMOVE_CANCELLED. These GUIDs are defined in Wdmguid.h.

`FileObject`

Pointer to a file object for the device.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION

HWPROFILE_CHANGE_NOTIFICATION

IoRegisterPlugPlayNotification

TRACE_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **TRACE_INFORMATION_CLASS** enumeration type is used to indicate types of information associated with a WMI event tracing session.

Syntax

```
typedef enum _TRACE_INFORMATION_CLASS {
  TraceIdClass,
  TraceHandleClass,
  TraceEnableFlagsClass,
  TraceEnableLevelClass,
  GlobalLoggerHandleClass,
  EventLoggerHandleClass,
  AllLoggerHandlesClass,
  TraceHandleByNameClass,
  LoggerEventsLostClass,
  TraceSessionSettingsClass,
  LoggerEventsLoggedClass,
  DiskIoNotifyRoutinesClass,
  TraceInformationClassReserved1,
  FltIoNotifyRoutinesClass,
  TraceInformationClassReserved2,
  WdfNotifyRoutinesClass,
  MaxTraceInformationClass
} TRACE_INFORMATION_CLASS;
```

Constants

TraceIdClass	Retrieves the logger ID (ULONG) of an event tracing session given a caller-supplied Wnode.
TraceHandleClass	Retrieves a trace handle (TRACEHANDLE) for an event tracing session given a caller-supplied logger ID (ULONG).
TraceEnableFlagsClass	Retrieves the enable flags (ULONG) set on a caller-supplied event trace handle (TRACEHANDLE).
TraceEnableLevelClass	Retrieves the enable level (ULONG) set on a caller-supplied event trace handle (TRACEHANDLE).
GlobalLoggerHandleClass	Retrieves an event trace handle (TRACEHANDLE) for the global logger.
EventLoggerHandleClass	Reserved for use by the operating system.
AllLoggerHandlesClass	Retrieves an array of event trace handles (TRACEHANDLE array) for all valid loggers.
TraceHandleByNameClass	Retrieves an event trace handle (TRACEHANDLE) identified by a caller-supplied friendly name (UNICODE_STRING structure in buffer).

LoggerEventsLostClass	Retrieves the number (ULONG) of events lost for a logger session given a caller-supplied logger ID (ULONG).
TraceSessionSettingsClass	Retrieves the settings (**ETW_TRACE_SESSION_SETTINGS** structure) for a logger session given a caller-supplied trace handle (TRACEHANDLE).
LoggerEventsLoggedClass	Retrieves the number (ULONG) of events logged in a logger session given a caller-supplied logger ID (ULONG).
DiskIoNotifyRoutinesClass	Reserved for use by the operating system.
TraceInformationClassReserved1	Reserved for use by the operating system.
FltIoNotifyRoutinesClass	Reserved for use by the operating system.
TraceInformationClassReserved2	Reserved for use by the operating system.
WdfNotifyRoutinesClass	Reserved for use by the operating system.
MaxTraceInformationClass	The maximum value in this enumeration type.

Remarks

TRACE_INFORMATION_CLASS is provided primarily for use with the WmiQueryTraceInformation routine, which returns information about a WMI event tracing session.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

WmiFireEvent

WmiQueryTraceInformation

WmiTraceMessage

WmiTraceMessageVa

TRANSACTION_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_BASIC_INFORMATION** structure contains information about a transaction object.

Syntax

```
typedef struct _TRANSACTION_BASIC_INFORMATION {
  GUID  TransactionId;
  ULONG State;
  ULONG Outcome;
} TRANSACTION_BASIC_INFORMATION, *PTRANSACTION_BASIC_INFORMATION;
```

Members

`TransactionId`

A GUID that KTM has assigned to the transaction object. This value is also known as the transaction's unit of work (UOW) identifier.

`State`

A TRANSACTION_STATE-typed value that specifies the transaction's current state.

`Outcome`

A TRANSACTION_OUTCOME-typed value that specifies the transaction's outcome (result).

Remarks

The **TRANSACTION_BASIC_INFORMATION** structure is used with the ZwQueryInformationTransaction routine. This routine fills in the structure's members.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_INFORMATION_CLASS

TRANSACTION_OUTCOME

TRANSACTION_STATE

TmGetTransactionId

ZwQueryInformationTransaction

TRANSACTION_ENLISTMENT_PAIR structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_ENLISTMENT_PAIR** structure contains information about an enlistment that is associated with a transaction object.

Syntax

```
typedef struct _TRANSACTION_ENLISTMENT_PAIR {
  GUID EnlistmentId;
  GUID ResourceManagerId;
} TRANSACTION_ENLISTMENT_PAIR, *PTRANSACTION_ENLISTMENT_PAIR;
```

Members

`EnlistmentId`

A GUID that KTM has assigned to the enlistment.

`ResourceManagerId`

A GUID that KTM has assigned to the resource manager that created the enlistment.

Remarks

The **TRANSACTION_ENLISTMENT_PAIR** structure is used as a member of the TRANSACTION_ENLISTMENTS_INFORMATION structure.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_ENLISTMENTS_INFORMATION

ZwQueryInformationTransaction

TRANSACTION_ENLISTMENTS_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_ENLISTMENTS_INFORMATION** structure contains information about the enlistments that are associated with a transaction object.

Syntax

```
typedef struct _TRANSACTION_ENLISTMENTS_INFORMATION {
  ULONG                      NumberOfEnlistments;
  TRANSACTION_ENLISTMENT_PAIR EnlistmentPair[1];
} TRANSACTION_ENLISTMENTS_INFORMATION, *PTRANSACTION_ENLISTMENTS_INFORMATION;
```

Members

`NumberOfEnlistments`

The number of enlistments that are associated with a transaction object. This is also the number of elements in the array that the **EnlistmentPair** member specifies.

`EnlistmentPair`

A caller-allocated array of TRANSACTION_ENLISTMENT_PAIR structures.

Remarks

The **TRANSACTION_ENLISTMENTS_INFORMATION** structure is used with the ZwQueryInformationTransaction routine. This routine fills in the structure's members.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_ENLISTMENT_PAIR

TRANSACTION_INFORMATION_CLASS

ZwQueryInformationTransaction

TRANSACTION_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_INFORMATION_CLASS** enumeration specifies the type of information that ZwSetInformationTransaction can set and ZwQueryInformationTransaction can retrieve for a transaction manager object.

Syntax

```
typedef enum _TRANSACTION_INFORMATION_CLASS {
  TransactionBasicInformation,
  TransactionPropertiesInformation,
  TransactionEnlistmentInformation,
  TransactionSuperiorEnlistmentInformation
} TRANSACTION_INFORMATION_CLASS;
```

Constants

TransactionBasicInformation	Information about a transaction manager object is stored in a TRANSACTION_BASIC_INFORMATION structure.
TransactionPropertiesInformation	Information about a transaction manager object is stored in a TRANSACTION_PROPERTIES_INFORMATION structure.
TransactionEnlistmentInformation	Information about a transaction manager object is stored in a TRANSACTION_ENLISTMENTS_INFORMATION structure.
TransactionSuperiorEnlistmentInformation	Information about a transaction manager object is stored in a **TRANSACTION_SUPERIOR_ENLISTMENT_INFORMATION** structure.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

ZwQueryInformationTransaction

ZwSetInformationTransaction

TRANSACTION_OUTCOME Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_OUTCOME** enumeration defines the outcomes (results) that KTM can assign to a transaction.

Syntax

```
typedef enum _TRANSACTION_OUTCOME {
  TransactionOutcomeUndetermined,
  TransactionOutcomeCommitted,
  TransactionOutcomeAborted
} TRANSACTION_OUTCOME;
```

Constants

TransactionOutcomeUndetermined	The transaction has not yet been committed or rolled back.
TransactionOutcomeCommitted	The transaction has been committed.
TransactionOutcomeAborted	The transaction has been rolled back.

Remarks

The **TRANSACTION_OUTCOME** enumeration is used in the TRANSACTION_BASIC_INFORMATION structure.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_BASIC_INFORMATION

TRANSACTION_PROPERTIES_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_PROPERTIES_INFORMATION** structure contains a transaction object's properties.

Syntax

```
typedef struct _TRANSACTION_PROPERTIES_INFORMATION {
  ULONG         IsolationLevel;
  ULONG         IsolationFlags;
  LARGE_INTEGER Timeout;
  ULONG         Outcome;
  ULONG         DescriptionLength;
  WCHAR         Description[1];
} TRANSACTION_PROPERTIES_INFORMATION, *PTRANSACTION_PROPERTIES_INFORMATION;
```

Members

`IsolationLevel`

Reserved for future use.

`IsolationFlags`

Reserved for future use.

`Timeout`

A time-out value. If the transaction has not been committed by the time specified by this parameter, KTM rolls back the transaction. The time-out value is expressed in system time units (100-nanosecond intervals), and can specify either an absolute time or a relative time. If the value of the **Timeout** member is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. Set this member to zero if no time-out is required. If **Timeout** is zero, the transaction never times out.

`Outcome`

A TRANSACTION_OUTCOME-typed value that identifies the transaction's outcome (result). You can obtain this value by calling ZwQueryInformationTransaction, but you cannot set the value with ZwSetInformationTransaction.

`DescriptionLength`

The size, in bytes, of the buffer that contains the **Description** member's array.

`Description`

A caller-allocated array of characters that contains or receives the transaction's description string. The TPS client can specify this string when it calls ZwCreateTransaction. The client can retrieve the string by calling by calling **ZwQueryInformationTransaction**, and it can modify the string by calling **ZwSetInformationTransaction**.

Remarks

The **TRANSACTION_PROPERTIES_INFORMATION** structure is used with the

ZwQueryInformationTransaction and ZwSetInformationTransaction routines.

Requirements

Minimum supported client	Available starting with Windows Vista.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_INFORMATION_CLASS

TRANSACTION_OUTCOME

ZwCreateTransaction

ZwQueryInformationTransaction

ZwSetInformationTransaction

TRANSACTION_STATE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTION_STATE** enumeration defines the states that KTM can assign to a transaction.

Syntax

```
typedef enum _TRANSACTION_STATE {
  TransactionStateNormal,
  TransactionStateIndoubt,
  TransactionStateCommittedNotify
} TRANSACTION_STATE;
```

Constants

TransactionStateNormal	The transaction's state is neither in doubt nor committed.
TransactionStateIndoubt	The transaction's state is in doubt (that is, KTM cannot determine whether the transaction should be committed or rolled back). A transaction that has been prepared enters the "in doubt" state if its superior transaction manager becomes unavailable.
TransactionStateCommittedNotify	The transaction has been committed. Commit notifications might (or might not) have been delivered to all enlistments.

Remarks

The **TRANSACTION_STATE** enumeration is used in the TRANSACTION_BASIC_INFORMATION structure.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTION_BASIC_INFORMATION

TRANSACTIONMANAGER_BASIC_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTIONMANAGER_BASIC_INFORMATION** structure contains information about a transaction manager object.

Syntax

```
typedef struct _TRANSACTIONMANAGER_BASIC_INFORMATION {
  GUID          TmIdentity;
  LARGE_INTEGER VirtualClock;
} TRANSACTIONMANAGER_BASIC_INFORMATION, *PTRANSACTIONMANAGER_BASIC_INFORMATION;
```

Members

`TmIdentity`

A GUID that KTM has assigned to a transaction manager object.

`VirtualClock`

The virtual clock value that is currently associated with a transaction manager object.

Remarks

The **TRANSACTIONMANAGER_BASIC_INFORMATION** structure is used with the ZwQueryInformationTransactionManager routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTIONMANAGER_INFORMATION_CLASS

ZwQueryInformationTransactionManager

TRANSACTIONMANAGER_INFORMATION_CLASS Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTIONMANAGER_INFORMATION_CLASS** enumeration specifies the type of information that the ZwQueryInformationTransactionManager routine can retrieve for a transaction manager object.

Syntax

```
typedef enum _TRANSACTIONMANAGER_INFORMATION_CLASS {
  TransactionManagerBasicInformation,
  TransactionManagerLogInformation,
  TransactionManagerLogPathInformation,
  TransactionManagerRecoveryInformation
} TRANSACTIONMANAGER_INFORMATION_CLASS;
```

Constants

TransactionManagerBasicInformation	Information about a transaction manager object is stored in a TRANSACTIONMANAGER_BASIC_INFORMATION structure.
TransactionManagerLogInformation	Information about a transaction manager object is stored in a TRANSACTIONMANAGER_LOG_INFORMATION structure.
TransactionManagerLogPathInformation	Information about a transaction manager object is stored in a TRANSACTIONMANAGER_LOGPATH_INFORMATION structure.
TransactionManagerRecoveryInformation	Information about a transaction manager object is stored in a TRANSACTIONMANAGER_RECOVERY_INFORMATION structure.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTIONMANAGER_BASIC_INFORMATION

TRANSACTIONMANAGER_LOGPATH_INFORMATION

TRANSACTIONMANAGER_LOG_INFORMATION

ZwQueryInformationTransactionManager

TRANSACTIONMANAGER_LOG_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTIONMANAGER_LOG_INFORMATION** structure contains information about a transaction manager object.

Syntax

```
typedef struct _TRANSACTIONMANAGER_LOG_INFORMATION {
  GUID LogIdentity;
} TRANSACTIONMANAGER_LOG_INFORMATION, *PTRANSACTIONMANAGER_LOG_INFORMATION;
```

Members

`LogIdentity`

A GUID that KTM uses to identify restart records in a transaction manager object's log stream. For more information about restart records, see Writing Restart Records to a CLFS Stream.

Remarks

The **TRANSACTIONMANAGER_LOG_INFORMATION** structure is used with the ZwQueryInformationTransactionManager routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTIONMANAGER_INFORMATION_CLASS

ZwQueryInformationTransactionManager

TRANSACTIONMANAGER_LOGPATH_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The **TRANSACTIONMANAGER_LOGPATH_INFORMATION** structure contains information about a transaction manager object.

Syntax

```
typedef struct _TRANSACTIONMANAGER_LOGPATH_INFORMATION {
  ULONG LogPathLength;
  WCHAR LogPath[1];
} TRANSACTIONMANAGER_LOGPATH_INFORMATION, *PTRANSACTIONMANAGER_LOGPATH_INFORMATION;
```

Members

`LogPathLength`

The number of elements in the **LogPath** member's character array.

`LogPath`

A caller-allocated character array that receives the path and file name of the log file stream that is associated with a transaction manager.

Remarks

The **TRANSACTIONMANAGER_LOGPATH_INFORMATION** structure is used with the ZwQueryInformationTransactionManager routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

TRANSACTIONMANAGER_INFORMATION_CLASS

ZwQueryInformationTransactionManager

TRANSACTIONMANAGER_RECOVERY_INFORMATION structure

1/30/2019 • 2 minutes to read • Edit Online

The TRANSACTIONMANAGER_RECOVERY_INFORMATION structure contains information about a transaction manager object.

Syntax

```
typedef struct _TRANSACTIONMANAGER_RECOVERY_INFORMATION {
  ULONGLONG LastRecoveredLsn;
} TRANSACTIONMANAGER_RECOVERY_INFORMATION, *PTRANSACTIONMANAGER_RECOVERY_INFORMATION;
```

Members

`LastRecoveredLsn`

The last log sequence number of the last log record that KTM obtained from CLFS during the most recent recovery operation.

Remarks

The TRANSACTIONMANAGER_RECOVERY_INFORMATION structure is used with the ZwQueryInformationTransactionManager routine.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Header	wdm.h (include Wdm.h, Ntifs.h)

See Also

TRANSACTIONMANAGER_INFORMATION_CLASS

ZwQueryInformationTransactionManager

WORK_QUEUE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **WORK_QUEUE_TYPE** enumeration type indicates the type of system worker thread that handles a work item.

Syntax

```
typedef enum _WORK_QUEUE_TYPE {
  CriticalWorkQueue,
  DelayedWorkQueue,
  HyperCriticalWorkQueue,
  NormalWorkQueue,
  BackgroundWorkQueue,
  RealTimeWorkQueue,
  SuperCriticalWorkQueue,
  MaximumWorkQueue,
  CustomPriorityWorkQueue
} WORK_QUEUE_TYPE;
```

Constants

CriticalWorkQueue	Indicates a real-time system worker thread. The assigned priority level is 13.
DelayedWorkQueue	Indicates an ordinary worker thread. The assigned priority level is 12.
HyperCriticalWorkQueue	System priority level. The assigned priority level is 15.
NormalWorkQueue	System priority level. The assigned priority level is 8
BackgroundWorkQueue	System priority level. The assigned priority level is 7.
RealTimeWorkQueue	System priority level. The assigned priority level is18.
SuperCriticalWorkQueue	System priority level. The assigned priority level is 14.
MaximumWorkQueue	System priority maximum. No priority level assigned.
CustomPriorityWorkQueue	The queue has a custom priority level assigned by the caller. The **CustomPriorityWorkQueue** value is the base priority level for the custom priority queue. Work items are queued at a particular priority by setting *QueueType* to **CustomPriorityWorkQueue** + *Priority* where *Priority* is the **KPRIORITY** value for the work item. This queue type is valid starting with Windows 8.1.

Remarks

For drivers targeting Windows 8.1 or later versions of Windows, use of only **CustomPriorityWorkQueue** with a

custom priority level is recommended.

A work queue priority < 17 is advised since queuing
at a higher priority may interfere with work item processing by critical system components.

For more information about system worker threads, see System Worker Threads.

Requirements

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

IoQueueWorkItem

IoQueueWorkItemEx

IoTryQueueWorkItem

ALLOCATE_FUNCTION_EX callback function

1/25/2019 • 4 minutes to read • Edit Online

The *LookasideListAllocateEx* routine allocates the storage for a new lookaside-list entry when a client requests an entry from a lookaside list that is empty.

Syntax

```
ALLOCATE_FUNCTION_EX AllocateFunctionEx;

PVOID AllocateFunctionEx(
  POOL_TYPE PoolType,
  SIZE_T NumberOfBytes,
  ULONG Tag,
  PLOOKASIDE_LIST_EX Lookaside
)
{...}
```

Parameters

`PoolType`

Specifies the type of storage to allocate for the new lookaside-list entry. The caller sets this parameter to a valid POOL_TYPE enumeration value, and possibly bitwise ORs this value with one of the following flag bits:

- POOL_RAISE_IF_ALLOCATION_FAILURE
- POOL_QUOTA_FAIL_INSTEAD_OF_RAISE

For more information about the POOL_RAISE_IF_ALLOCATION_FAILURE flag, see ExAllocatePoolWithTag. For more information about the POOL_QUOTA_FAIL_INSTEAD_OF_RAISE flag, see ExAllocatePoolWithQuotaTag.

If, in the ExInitializeLookasideListEx call that initialized the lookaside list, the *Flags* parameter is zero, the *PoolType* parameter that the *LookasideListAllocateEx* routine receives is the same *PoolType* parameter value that was passed to **ExInitializeLookasideListEx**.

If, in the **ExInitializeLookasideListEx** call, *Flags* = EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL, the *PoolType* parameter that the *LookasideListAllocateEx* routine receives is the bitwise OR of POOL_RAISE_IF_ALLOCATION_FAILURE and the *PoolType* parameter value that was passed to **ExInitializeLookasideListEx**. The *LookasideListAllocateEx* routine can pass its *PoolType* parameter value, without modification, to the **ExAllocatePoolWithTag** routine.

If, in the **ExInitializeLookasideListEx** call, Flags = EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE, the *PoolType* parameter that the *LookasideListAllocateEx* routine receives is the bitwise OR of POOL_QUOTA_FAIL_INSTEAD_OF_RAISE and the *PoolType* value that was passed to **ExInitializeLookasideListEx**. The *LookasideListAllocateEx* routine can pass its *PoolType* parameter value, without modification, to the **ExAllocatePoolWithQuotaTag** routine.

`NumberOfBytes`

Specifies the size, in bytes, of the lookaside-list entry to allocate.

`Tag`

Specifies the four-byte pool tag to use to mark the allocated storage for the new lookaside-list entry. For more information about pool tags, see the description of the *Tag* parameter in ExAllocatePoolWithTag.

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes the lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

Return Value

LookasideListAllocateEx returns a pointer to the allocated lookaside-list entry. If the routine cannot allocate an entry, it returns **NULL**.

Remarks

A driver that creates a lookaside list can implement a *LookasideListAllocateEx* routine to dynamically allocate buffers for the list. A buffer that is not in use is stored as an entry in the list. All entries in a lookaside list are buffers of a uniform size, which the driver specifies when the list is initialized.

The driver supplies a pointer to a custom *LookasideListAllocateEx* routine as an input parameter in the **ExInitializeLookasideListEx** call that initializes the lookaside list. If the driver sets this parameter to **NULL**, the lookaside list uses a default allocation routine instead.

A driver calls the ExAllocateFromLookasideListEx routine to allocate an entry from a lookaside list. If the list is empty (contains no entries), **ExAllocateFromLookasideListEx** calls *LookasideListAllocateEx* to dynamically allocate the storage for a new entry. *LookasideListAllocateEx* returns a pointer to the newly allocated entry if the allocation is successful. Otherwise, it returns **NULL**.

The *PoolType*, *NumberOfBytes*, *Tag*, and *Lookaside* parameters contain the same values that were passed as input parameters in the **ExInitializeLookasideListEx** call that initialized the lookaside list.

The *LookasideListAllocateEx* routine can use the *Lookaside* parameter to access private context data that the driver has associated with the lookaside list. For more information, see the code example in ExInitializeLookasideListEx.

For more information about lookaside lists, see Using Lookaside Lists.

The *LookasideListAllocateEx* routine is called at the same IRQL as the call to **ExAllocateFromLookasideListEx** that requests the entry. For a call that requests an entry that resides in paged memory, the caller must be running IRQL <= APC_LEVEL. For a call that requests an entry that resides in nonpaged memory, the caller must be running IRQL <= DISPATCH_LEVEL.

Examples

To define a *LookasideListAllocateEx* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *LookasideListAllocateEx* callback routine that is named `MyLookasideListAllocateEx`, use the FREE_FUNCTION_EX type as shown in this code example:

```
FREE_FUNCTION_EX MyLookasideListFreeEx;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyLookasideListFreeEx(
    PVOID  Buffer,
    PLOOKASIDE_LIST_EX  Lookaside
    )
  {
      // Function body
  }
```

The FREE_FUNCTION_EX function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the FREE_FUNCTION_EX function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows Vista and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	See Remarks section.

See Also

ExAllocateFromLookasideListEx

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExInitializeLookasideListEx

LOOKASIDE_LIST_EX

POOL_TYPE

AppendTailList function

1/25/2019 • 3 minutes to read • Edit Online

The **AppendTailList** routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of **LIST_ENTRY** structures.

Syntax

```
void AppendTailList(
  PLIST_ENTRY ListHead,
  PLIST_ENTRY ListToAppend
);
```

Parameters

`ListHead`

A pointer to the head of the list to which the list pointed to by *ListToAppend* is to be appended. This parameter must be a valid, non-NULL pointer to the LIST_ENTRY structure that is the head of a list. This list can be empty.

`ListToAppend`

A pointer to the first entry in the list to append to the list pointed to by *ListHead*. This parameter must be a valid, non-NULL pointer to the LIST_ENTRY structure that is the first entry in a headless list. For more information, see Remarks.

Return Value

None.

Remarks

This routine constructs a list that contains all the combined entries from the input *ListHead* and *ListToAppend* lists. When the routine returns, the LIST_ENTRY structure pointed to by the *ListHead* parameter is the head of the resulting combined list. The **LIST_ENTRY** structure pointed to by the *ListToAppend* parameter is an entry in the resulting combined list.

The caller sets the *ListToAppend* parameter to point to the first entry in the list to be appended. In this regard, *AppendTailList* does not conform to the convention followed by the other doubly linked list routines that use the **LIST_ENTRY** structure. In these other routines, a parameter that represents a list points to the list head instead of to the first entry in the list. If your driver code uses a list head to represent the list to be appended, the list head must be removed from this list before the list is passed as the *ListToAppend* parameter to **AppendTailList**. For a code example that shows how to remove this list head, see Examples.

AppendTailList can be used to insert a single entry at the tail of a list, but the **Flink** and **Blink** members of the **LIST_ENTRY** structure pointed to by *ListToAppend* must first be initialized—typically, by calling the InitializeListHead routine. In contrast, the InsertTailList routine does not require the **Flink** and **Blink** members of the **LIST_ENTRY** structure pointed to by the *Entry* parameter to be initialized before the **InsertTailList** call. For this reason, **InsertTailList** might be more convenient to use to insert a single entry at the tail of a list.

A driver can access the **Flink** and **Blink** members of a **LIST_ENTRY**, but the members must only be updated by

the system routines supplied for this purpose.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **AppendTailList** can be running at any IRQL. If **AppendTailList** is called at IRQL >= DISPATCH_LEVEL, the storage for the list entries must be memory-resident.

Examples

The following code example shows how to write a function named `MyAppendTailList` that is similar to **AppendTailList**, but that treats the *ListToAppend* parameter as a pointer to a list head instead of as a pointer to the first entry in a (headless) list. Unlike **AppendTailList**, the `MyAppendTailList` function avoids including the **LIST_ENTRY** structure pointed to by the *ListToAppend* parameter in the entries that are appended to the list pointed to by the *ListHead* parameter.

```
VOID MyAppendTailList(
    _Inout_  PLIST_ENTRY ListHead,
    _Inout_  PLIST_ENTRY ListToAppend
    )
{
    PLIST_ENTRY entry = ListToAppend->Flink;

    if (!IsListEmpty(ListToAppend))
    {
        RemoveEntryList(ListToAppend);
        InitializeListHead(ListToAppend);
        AppendTailList(ListHead, entry);
    }
}
```

The `MyAppendTailList` function in this code example treats both the *ListHead* and *ListToAppend* parameters as pointers to list heads. When this function returns, the list pointed to by *ListToAppend* is empty; that is, it consists of a list head that has no associated list entries. All of the entries that were initially in this list have been appended to the list pointed to by *ListHead*.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

InitializeListHead

InsertTailList

LIST_ENTRY

RemoveEntryList

BOUND_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *BoundCallback* routine is executed whenever the system issues a bounds exception for a user-mode thread.

Syntax

```
BOUND_CALLBACK BoundCallback;

BOUND_CALLBACK_STATUS BoundCallback(

)
{...}
```

Parameters

This callback function has no parameters.

Return Value

A BOUND_CALLBACK_STATUS value that indicates how the bounds exception was processed by the callback function.

Remarks

Drivers can supply a *BoundCallback* that is called when the system issues a bounds exception.

Use KeRegisterBoundCallback to register a *BoundCallback* routine. A driver can subsequently remove the callback by using the KeDeregisterBoundCallback routine. If the driver can be unloaded, it must remove any registered callbacks in its Unload routine.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Windows
Header	wdm.h

See Also

KeDeregisterBoundCallback

KeRegisterBoundCallback

ClfsAddLogContainer function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsAddLogContainer** routine adds a container to a CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsAddLogContainer(
  PLOG_FILE_OBJECT plfoLog,
  PULONGLONG       pcbContainer,
  PUNICODE_STRING  puszContainerPath
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents the log to which the container will be added. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`pcbContainer`

A pointer to a ULONGLONG-typed variable. The role of this parameter depends on whether the log currently has at least one container.

- If the log currently has no containers, the caller supplies a positive integer that is the requested size, in bytes, of the new container. CLFS creates a container that is the requested size rounded up to a multiple of 512KB (for a dedicated log) or 1MB (for a multiplexed log).
- If the log currently has at least one container and this parameter is the **NULL** pointer, CLFS creates a new container that is the same size as the existing containers. (All containers in a log must be the same size.)
- If the log currently has at least one container and this parameter is a valid pointer, the caller supplies a positive integer that is the requested size, in bytes, of the new container. The requested size is rounded up to a multiple of 512KB (for a dedicated log) or 1MB (for a multiplexed log). If the rounded-up size is at least as large as the existing container size, CLFS creates a new container that is the same size as the existing containers. On successful return, this parameter receives the existing container size. If the rounded-up size is less than the existing container size, **ClfsAddLogContainer** fails.

`puszContainerPath`

A pointer to a UNICODE_STRING structure that supplies the path name for the new container. The path can be absolute or relative to the location of the base log file. Paths that are relative to the base log file must begin with CLFS_CONTAINER_RELATIVE_PREFIX, which is the string literal (L"%BLF%\").

Return Value

ClfsAddLogContainer returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

A container is a contiguous extent on stable storage. For example, a container could be a contiguous file on disk. A

log is a set of containers along with a base log file. For more information about containers, see CLFS Stable Storage.

A container specified by a path that is relative to the base log file must be in the same directory as the base log file or a subdirectory of the base log file. The directories "." and ".." are not allowed in a relative path.

A log can have some containers with absolute paths and other containers with relative paths.

Containers are created in a non-compressed mode and are initialized with zeros.

A log must have at least two containers before any I/O can be performed on it.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainerSet

ClfsRemoveLogContainer

ClfsAddLogContainerSet function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsAddLogContainerSet** routine atomically adds a set of containers to a CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsAddLogContainerSet(
  PLOG_FILE_OBJECT plfoLog,
  USHORT           cContainers,
  PULONGLONG       pcbContainer,
  PUNICODE_STRING  rguszContainerPath
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents the log to which the containers will be added. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`cContainers`

The number of containers in the set. This is the number of elements in the *rguszContainerPath* array.

`pcbContainer`

A pointer to a ULONGLONG-typed variable. The role of this parameter depends on whether the log currently has at least one container.

- If the log currently has no containers, the caller supplies a positive integer that is the requested size, in bytes, of an individual container. CLFS creates containers that are the requested size rounded up to a multiple of 512KB (for a dedicated log) or 1MB (for a multiplexed log).
- If the log currently has at least one container and this parameter is the **NULL** pointer, CLFS creates new containers that are the same size as the existing containers. (All containers in a log must be the same size.)
- If the log currently has at least one container and this parameter is a valid pointer, the caller supplies a positive integer that is the requested size, in bytes, of an individual container. The requested size is rounded up to a multiple of 512KB (for a dedicated log) or 1MB (for a shared log). If the rounded-up size is at least as large as the existing container size, CLFS creates new containers that are the same size as the existing containers. On successful return, this parameter receives the existing container size. If the rounded-up size is less than the existing container size, **ClfsAddLogContainerSet** fails.

`rguszContainerPath`

A pointer to an array of UNICODE_STRING structures. Each string supplies the path name for one of the new containers. The number of elements in the array is given by *cContainers*. A given path can be absolute or relative to the location of the base log file. Paths that are relative to the base log file must begin with CLFS_CONTAINER_RELATIVE_PREFIX, which is the string literal (L"%BLF%\").

Return Value

ClfsAddLogContainer returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes

defined in Ntstatus.h.

Remarks

A container is a contiguous extent on stable storage. For example, a container could be a contiguous file on disk. A log is a set of containers along with a base log file. For more information about containers, see CLFS Stable Storage.

A container specified by a path that is relative to the base log file must be in the same directory as the base log file or a subdirectory of the base log file. The directories "." and ".." are not allowed in a relative path.

A log can have some containers with absolute paths and other containers with relative paths.

Containers are created in a non-compressed mode and are initialized with zeros.

A log must have at least two containers before any I/O can be performed on it.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainer

ClfsRemoveLogContainerSet

ClfsAdvanceLogBase function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsAdvanceLogBase** routine sets the base LSN of a CLFS stream.

Syntax

```
CLFSUSER_API NTSTATUS ClfsAdvanceLogBase(
  PVOID     pvMarshalContext,
  PCLFS_LSN plsnBase,
  ULONG     fFlags
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`plsnBase`

A pointer to a CLFS_LSN structure that contains the new base LSN. This parameter must be the LSN of one of the records in the stream. Also, this parameter must be greater than or equal to the stream's current base LSN and less than or equal to the stream's current last LSN.

`fFlags`

This parameter is reserved for system use. Callers must set this parameter to zero.

Return Value

ClfsAdvanceLogBase returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

ClfsAdvanceLogBase does not write any records to the CLFS log; the only updates to the log are in the metadata. If you want to update the base LSN and write a restart record to a stream at the same time, call ClfsWriteRestartArea.

Whenever possible, CLFS avoids writing queued log records that have LSNs less than the new base LSN to stable storage.

ClfsAdvanceLogBase does not check to see whether the LSN supplied in *plsnBase* is actually the LSN of one of the records in the stream. If the caller sets *plsnBase* to an LSN that is not the LSN of one of the records in the stream, the stream's base LSN will be set to a meaningless value.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsWriteRestartArea

ClfsAlignReservedLog function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsAlignReservedLog** routine calculates the size of the space that must be reserved for a specified set of records. The size calculation includes the space required for headers and the space required for sector alignment.

Syntax

```
CLFSUSER_API NTSTATUS ClfsAlignReservedLog(
  PVOID       pvMarshalContext,
  ULONG       cRecords,
  LONGLONG [] rgcbReservation,
  PLONGLONG   pcbAlignReservation
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`cRecords`

The number of elements in the array pointed to by *rgcbReservation*.

`rgcbReservation`

A pointer to an array of LONGLONG-typed variables that supplies a list of record sizes (in bytes).

`pcbAlignReservation`

A pointer to a LONGLONG-typed variable that receives the size, in bytes, of the required reservation.

Return Value

ClfsAlignReservedLog returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAllocReservedLog

ClfsAllocReservedLog function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsAllocReservedLog** routine reserves space in a marshalling area for a set of records.

Syntax

```
CLFSUSER_API NTSTATUS ClfsAllocReservedLog(
  PVOID     pvMarshalContext,
  ULONG     cRecords,
  PLONGLONG pcbAdjustment
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`cRecords`

The number of records in the set. This value must match the value of *cRecords* previously passed to ClfsAlignReservedLog.

`pcbAdjustment`

A pointer to a LONGLONG-typed variable that supplies the size, in bytes, of the reservation that will hold all the records in the set. The caller previously obtained this size by calling ClfsAlignReservedLog.

Return Value

ClfsAllocReservedLog returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAlignReservedLog

ClfsCloseAndResetLogFile function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsCloseAndResetLogFile** routine releases all references to a specified log file object and marks its associated stream for reset.

Syntax

```
CLFSUSER_API NTSTATUS ClfsCloseAndResetLogFile(
  PLOG_FILE_OBJECT plfoLog
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure for which references will be released. The caller previously obtained this pointer by calling ClfsCreateLogFile.

Return Value

ClfsCloseAndResetLogFile returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

ClfsCloseAndResetLogFile releases all references, acquired by CLFS, to the specified log file object. Typically, the client will not have acquired any additional references to the log file object. However, if the client has obtained additional refereces, it is the client's responsibility to release them.

ClfsCloseAndResetLogFile causes the specified log file object's reference count to drop to zero, at which time the log file object is closed (that is, its memory is deallocated). The stream represented by the log file object might not be reset, however, because there could be other log file objects that represent the same stream. The stream is reset after all log file objects that represent the stream are closed (for example by calls to ClfsCloseLogFileObject).

A reset stream is like a newly created stream that has never had any records written to it.

On return from this routine, *plfoLog* is invalid.

You can call **ClfsCloseAndResetLogFile** at most 124 times for a given stream.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop

Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCloseLogFileObject

ClfsDeleteLogByPointer

ClfsDeleteLogFile

ClfsCloseLogFileObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsCloseLogFileObject** routine releases all references to a log file object.

Syntax

```
CLFSUSER_API NTSTATUS ClfsCloseLogFileObject(
  PLOG_FILE_OBJECT plfoLog
);
```

Parameters

`plfoLog`

A pointer to the LOG_FILE_OBJECT structure for which references will be released.The caller previously obtained this pointer by calling ClfsCreateLogFile.

Return Value

ClfsCloseLogFileObject returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

ClfsCloseLogFileObject releases all references, acquired by CLFS, to the specified log file object. Typically, the client will not have acquired any additional references to the log file object. However, if the client has obtained additional refereces, it is the client's responsibility to release them.

ClfsCloseLogFileObject causes the specified log file object's reference count to drop to zero, at which time the log file object is closed (that is, its memory is deallocated). The stream represented by the log file object might remain open, however, because there could be other log file objects that represent the same stream. The stream is closed after all log file objects that represent the stream are closed.

On return from this routine, *plfoLog* is invalid.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCloseAndResetLogFile

ClfsCreateLogFile

ClfsCreateLogFile function

1/25/2019 • 6 minutes to read • Edit Online

The **ClfsCreateLogFile** routine creates or opens a CLFS stream. If necessary, **ClfsCreateLogFile** also creates the underlying physical log that holds the stream's records.

Syntax

```
CLFSUSER_API NTSTATUS ClfsCreateLogFile(
  PPLOG_FILE_OBJECT    pplfoLog,
  PUNICODE_STRING      puszLogFileName,
  ACCESS_MASK          fDesiredAccess,
  ULONG                dwShareMode,
  PSECURITY_DESCRIPTOR psdLogFile,
  ULONG                fCreateDisposition,
  ULONG                fCreateOptions,
  ULONG                fFlagsAndAttributes,
  ULONG                fLogOptionFlag,
  PVOID                pvContext,
  ULONG                cbContext
);
```

Parameters

`pplfoLog`

A pointer to a variable that receives a pointer to a LOG_FILE_OBJECT structure that represents an open instance of the stream.

`puszLogFileName`

A pointer to a UNICODE_STRING structure that supplies the name of the stream or the underlying physical log.

If the stream already exists and is the only stream of a dedicated log, the name has the form log:*physical log name*, where *physical log name* is the path name, on the underlying file system, of the existing physical log that contains the stream's records.

If the stream does not already exist and is to become the only stream of a dedicated log (that does not yet exist), the name has the form log:*physical log name*, where *physical log name* is the path name, on the underlying file system, of the physical log that will be created to hold the stream's records.

If the stream is (or is to become) one of the streams of a multiplexed log, the name has the form log:*physical log name*::*stream name*, where *physical log name* is the path name, on the underlying file system, of the physical log that holds the stream's records, and *stream name* is the name of a stream that shares (or will share) that physical log.

If you want to create a multiplexed log that has no streams for the moment, use a name of the form log:*physical log name*::, where *physical log name* is the path name, on the underlying file system, of the physical log to be created.

The following list gives some examples of valid names.

- "Log:c:\myLog" creates or opens a dedicated log and its one stream.
- "Log:c:\myCommonLog::" creates a multiplexed log that does not yet have any streams.

- "Log:c:\myCommonLog::Stream1" creates or opens one of the streams (Stream1) of a multiplexed log.

`fDesiredAccess`

An ACCESS_MASK that supplies the type of access the client will have (by using the pointer returned in *pplfoLog*) to the stream. If this parameter is zero, clients can query the stream for its attributes, but cannot read from or write to the stream. This parameter can be zero or any combination of the following flags:

FLAG	MEANING
GENERIC_READ	The client has read access to the stream.
GENERIC_WRITE	The client has write access to the stream.
DELETE	The client can mark the stream for deletion.

`dwShareMode`

The sharing mode of the stream, which can be zero (not shared) or any combination of the following flags:

FLAG	MEANING
FILE_SHARE_DELETE	Subsequent requests to open the stream with delete access will succeed.
FILE_SHARE_READ	Subsequent requests to open the stream with read access will succeed.
FILE_SHARE_WRITE	Subsequent requests to open the stream with write access will succeed.

`psdLogFile`

A pointer to a SECURITY_DESCRIPTOR structure that supplies security attributes for the stream. This parameter can be **NULL**.

`fCreateDisposition`

The action to take that depends on whether the stream already exists. This parameter must be set to one of the following values:

VALUE	MEANING
CREATE_NEW	Create a new stream if the stream does not already exits. Fail if the stream already exists.
OPEN_EXISTING	Open an existing stream. Fail if the stream does not already exist.
OPEN_ALWAYS	Open an existing stream. Create the stream if it does not already exist.

`fCreateOptions`

A set of flags that specify options to apply when creating or opening the stream. This parameter can be zero or a compatible combination of the following flags:

FLAG	MEANING
FILE_NO_INTERMEDIATE_BUFFERING	The stream's records cannot be cached in a driver's internal buffers.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the stream are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. If this flag is set, the FILE_SYNCHRONOUS_IO_NONALERT flag must be cleared.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the stream are performed synchronously. Waits in the system that synchronize I/O queuing and completion are not subject to alerts. If this flag is set, the FILE_SYNCHRONOUS_IO_ALERT flag must be cleared.

`fFlagsAndAttributes`

A value that specifies whether the stream is opened for normal or read-only access. This parameter must be set to either

FILE_ATTRIBUTE_NORMAL or FILE_ATTRIBUTE_READONLY.

`fLogOptionFlag`

A hint about the relationship between CLFS and the component creating or opening the stream. This parameter must be set to one of the following values:

VALUE	MEANING
CLFS_FLAG_NO_FLAGS	CLFS and the creating component have the standard, normal relationship. Kernel-mode components use this value unless they fall into one of the three other categories listed in this table. If *pvContext* is not **NULL**, CLFS verifies that *cbContext* is greater than zero. Otherwise, *pvContext* and *cbContext* are ignored.
CLFS_FLAG_REENTRANT_FILE_SYSTEM	The creating component is the file system that provides the underlying storage for CLFS. CLFS uses the file system for allocating containers, and the file system uses CLFS streams. In this case, it is possible for the file system to call CLFS and for CLFS to make calls back into the file system on the same thread or different threads. If *pvContext* is not **NULL**, CLFS verifies that *cbContext* is greater than zero. Otherwise, *pvContext* and *cbContext* are ignored.
CLFS_FLAG_NON_REENTRANT_FILTER	The creating component is a file system filter driver that sends all of its CLFS I/O to a specified level below itself on the filter stack. This option allows a filter driver to create a CLFS log without seeing its own logging I/O. The caller passes the non-**NULL** target device object in the *pvContext* parameter with *cbContext* set to the appropriate size. CLFS uses the IoCreateFileSpecifyDeviceObjectHint routine to create containers at a targeted level in the I/O filter stack specified by the device object.

CLFS_FLAG_REENTRANT_FILTER	The creating component is a file system filter driver that sends all of its CLFS I/O to the top of the filter stack. The filter has a recursive relationship with CLFS because it filters its own logging I/O when CLFS performs any file system operation on its containers. The *pvContext* parameter provides a means for filters to associate a recognizable context with its CLFS containers as log I/O comes down the filter stack. The *cbContext* parameter specifies the size of the opaque context in bytes.
CLFS_FLAG_MINIFILTER_LEVEL	The creating component is a file system minifilter driver that sends all of its CLFS I/O to a specified level below itself on the filter stack. This option allows a minifilter to create a CLFS log without seeing its own logging I/O. The caller passes the non-**NULL** minifilter context object in the *pvContext* parameter with *cbContext* set to the appropriate size. CLFS uses the **IoCreateFileSpecifyDeviceObjectHint** routine to create containers at an altitude (specified within the minifilter context) in the filter manager's minifilter stack.

`pvContext`

A pointer to a context. The way the context is interpreted depends on the value passed to *fLogOptionsFlag*.

`cbContext`

The size, in bytes, of the context pointed to by *pvContext*. If *pvContext* is not **NULL**, this parameter must be greater than zero.

Return Value

ClfsCreateLogFile returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

When you create a CLFS stream, it is backed by an underlying physical CLFS log. The underlying log can be either dedicated (backs only one stream) or multiplexed (backs several streams). A dedicated log cannot be converted to a multiplexed log, and a multiplexed log cannot be converted to a dedicated log.

A physical CLFS log name does not include the .blf extension.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCloseAndResetLogFile

ClfsCloseLogFileObject

ClfsDeleteLogByPointer

ClfsDeleteLogFile

ClfsCreateMarshallingArea function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsCreateMarshallingArea** routine creates a marshalling area for a CLFS stream and returns a pointer to an opaque context that represents the new marshalling area.

Syntax

```
CLFSUSER_API NTSTATUS ClfsCreateMarshallingArea(
  PLOG_FILE_OBJECT   plfoLog,
  POOL_TYPE          ePoolType,
  PALLOCATE_FUNCTION pfnAllocBuffer,
  PFREE_FUNCTION     pfnFreeBuffer,
  ULONG              cbMarshallingBuffer,
  ULONG              cMaxWriteBuffers,
  ULONG              cMaxReadBuffers,
  PVOID              *ppvMarshalContext
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`ePoolType`

A POOL_TYPE value that specifies the type of memory (paged, non-paged, for example) that the new marshalling area will use for its log I/O blocks.

`pfnAllocBuffer`

Either **NULL** or a pointer to a caller-supplied function that allocates a log I/O block for the marshalling area. The allocation function has the following prototype:

```
PVOID
(*PALLOCATE_FUNCTION) (
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
    IN ULONG Tag
    );
```

The return value of the allocation function is a pointer to the newly allocated log I/O block. `pfnFreeBuffer` Either **NULL** or a pointer to a caller-supplied function that frees a log I/O block that was previously allocated by *pfnAllocBuffer*. The function has the following prototype:

```
VOID
(*PFREE_FUNCTION) (
    IN PVOID Buffer
    );
```

`cbMarshallingBuffer`

The size, in bytes, of the individual log I/O blocks that the new marshalling area uses. This must be a multiple of the sector size on the stable storage medium. The sector size is the *lpBytesPerSector* value returned from **GetDiskFreeSpace**.

`cMaxWriteBuffers`

The maximum number of I/O blocks that can be allocated at one time for write operations. This parameter affects the frequency of data flushes. If you do not need to control the frequency of data flushes, set this parameter to INFINITE.

`cMaxReadBuffers`

The maximum number of log I/O blocks that can be allocated at one time for read operations.

`ppvMarshalContext`

A pointer to a variable that receives a pointer to an opaque context that represents the new marshalling area.

Return Value

ClfsCreateMarshallingArea returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The *pfnAllocBuffer* and *pfnFreeBuffer* parameters must both point to caller-allocated functions, or they must both be **NULL**. If they are both **NULL**, CLFS provides default functions for allocating and freeing log I/O blocks.

Before calling **ClfsCreateMarshallingArea**, you must add at least two containers to the underlying log by calling ClfsAddLogContainer or ClfsAddLogContainerSet.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys

IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainer

ClfsAddLogContainerSet

ClfsCreateLogFile

ClfsDeleteMarshallingArea

LOG_FILE_OBJECT

POOL_TYPE

ClfsCreateScanContext function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsCreateScanContext** routine creates a scan context that can be used to iterate over the containers of a specified CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsCreateScanContext(
  PLOG_FILE_OBJECT   plfoLog,
  ULONG              cFromContainer,
  ULONG              cContainers,
  CLFS_SCAN_MODE     eScanMode,
  PCLFS_SCAN_CONTEXT pcxScan
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. The scan context is created for the log that provides the underlying storage for that stream. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`cFromContainer`

The index of the first container to be scanned. Containers are indexed starting at zero.

`cContainers`

The number of containers to be scanned with each call to ClfsScanLogContainers.

`eScanMode`

A set of flags that specify whether the scan context is set up for scanning forward or backward and whether the scan context should be reinitialized. The following three flags are available for callers of this routine.

VALUE	MEANING
CLFS_SCAN_FORWARD	Set up the scan context for scanning in the forward direction. For example, if container 5 is the most recently scanned container and *cContainers* is 2, then a forward scan would return descriptors for containers 6 and 7.
CLFS_SCAN_BACKWARD	Set up the scan context for scanning in the backward direction. For example, if container 5 is the most recently scanned container and *cContainers* is 2, then a backward scan would return descriptors for containers 4 and 3.
CLFS_SCAN_INIT	Reinitialize the scan context. The next time **ClfsScanLogContainers** is called, it will behave as if it were being called for the first time after the creation of the scan context.

If *pcxScan* points to a CLFS_SCAN_CONTEXT structure that is being passed to this routine for the first time, one of the direction flags (CLFS_SCAN_FORWARD or CLFS_SCAN_BACKWARD) must be set. The CLFS_SCAN_INIT flag must not be set.

If *pcxScan* points to a CLFS_SCAN_CONTEXT structure that has been passed to this routine previously, the CLFS_SCAN_INIT flag must be set. Also, one and only one of the direction flags (CLFS_SCAN_FORWARD or CLFS_SCAN_BACKWARD) must be set.

`pcxScan`

A pointer to a caller-allocated CLFS_SCAN_CONTEXT structure whose members are initialized by this routine. This structure is later passed to ClfsScanLogContainers.

Return Value

ClfsCreateScanContext returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsScanLogContainers

ClfsDeleteLogByPointer function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsDeleteLogByPointer** routine marks a CLFS stream for deletion.

Syntax

```
CLFSUSER_API NTSTATUS ClfsDeleteLogByPointer(
  PLOG_FILE_OBJECT plfoLog
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents an open instance of the stream to be deleted. The caller previously obtained this pointer by calling ClfsCreateLogFile.

Return Value

ClfsDeleteLogByPointer returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

ClfsDeleteLogByPointer marks a stream for deletion but does not close the log file object pointed to by *plfoLog*. To close a log file object, call ClfsCloseLogFileObject. A stream marked for deletion is deleted after all log file objects associated with the stream are closed.

A CLFS stream marked for deletion will refuse subsequent requests to open the stream.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCreateLogFile

ClfsDeleteLogFile

ClfsDeleteLogFile function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsDeleteLogFile** routine marks a CLFS stream for deletion.

Syntax

```
CLFSUSER_API NTSTATUS ClfsDeleteLogFile(
  PUNICODE_STRING puszLogFileName,
  PVOID           pvReserved,
  ULONG           fLogOptionFlag,
  PVOID           pvContext,
  ULONG           cbContext
);
```

Parameters

`puszLogFileName`

A pointer to a UNICODE_STRING structure that supplies the name of the CLFS stream to be deleted.

If the stream to be deleted is the only stream of a dedicated log, the name has the form log:*physical log name*, where *physical log name* is the path name of the physical log on the underlying file system.

If the stream to be deleted is one of the streams of a multiplexed log, the name has the form log:*physical log name*::*stream name*, where *physical log name* is the path name of the physical log on the underlying file system and *stream name* is the unique name of the stream to be deleted.

`pvReserved`

Reserved.

`fLogOptionFlag`

A value that indicates the relationship between CLFS and the component that is deleting the log. For a list of possible values, see the description of the *fLogOptionFlag* parameter of the **ClfsCreateLogFile** routine.

`pvContext`

A pointer to a context. The way the context is interpreted depends on the value passed in *fLogOptionFlag*.

`cbContext`

The size, in bytes, of the context pointed to by *pvContext*. If *pvContext* is not **NULL**, this parameter must be greater than zero.

Return Value

ClfsDeleteLogFile returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The Common Log File System (CLFS) uses the LOG_FILE_OBJECT structure to represent logs. The

ClfsCreateLogFile function returns a pointer to LOG_FILE_OBJECT, which clients then pass to other CLFS functions.

CLFS clients do not directly access the members of a LOG_FILE_OBJECT structure. For information about the members, see FILE_OBJECT.

ClfsDeleteLogFile marks a stream for deletion but does not close any log file objects that are currently open. To close a log file object, call ClfsCloseLogFileObject. A stream marked for deletion is deleted after all log file objects associated with the stream are closed.

A CLFS stream marked for deletion will refuse subsequent requests to open the stream.

The name of a physical CLFS log does not include the .blf extension.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCloseAndResetLogFile

ClfsCloseLogFileObject

ClfsCreateLogFile

ClfsDeleteLogByPointer

ClfsDeleteMarshallingArea function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsDeleteMarshallingArea** routine deletes a marshalling area.

Syntax

```
CLFSUSER_API NTSTATUS ClfsDeleteMarshallingArea(
  PVOID pvMarshalContext
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

Return Value

ClfsDeleteMarshallingArea returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

Before deleting the marshalling area, **ClfsDeleteMarshallingArea** flushes all of the log I/O blocks associated with the marshalling area.

The marshalling area goes away only after all references to it go away. For example, the marshalling area remains in memory if it still has an open read context.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCreateMarshallingArea

ClfsFlushBuffers function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsFlushBuffers** routine forces all log I/O blocks in a specified marshalling area to stable storage.

Syntax

```
CLFSUSER_API NTSTATUS ClfsFlushBuffers(
  PVOID pvMarshalContext
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

Return Value

ClfsFlushBuffers returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

Calling **ClfsFlushBuffers** is equivalent to calling ClfsFlushToLsn with the *plsnFlush* parameter set to CLFS_LSN_NULL.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsFlushToLsn

ClfsFlushToLsn function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsFlushToLsn** routine forces, to stable storage, all records that have an LSN less than or equal to a specified LSN.

Syntax

```
CLFSUSER_API NTSTATUS ClfsFlushToLsn(
  PVOID     pvMarshalContext,
  PCLFS_LSN plsnFlush,
  PCLFS_LSN plsnLastFlushed
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context associated with a marshalling area. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`plsnFlush`

A pointer to a CLFS_LSN structure. All records that have an LSN less than or equal to *plsnFlush* are guaranteed to be forced to stable storage. If *plsnFlush* is equal to CLFS_LSN_NULL, then all records in the marshalling area are forced to stable storage.

`plsnLastFlushed`

A pointer to a **CLFS_LSN** structure that receives the LSN of the oldest record that was not flushed. This is the LSN of the record immediately following the last record flushed.

> **Note** On successful return, *plsnLastFlushed* is greater than the value supplied in *plsnFlush*. However, *plsnLastFlushed* does not necessarily point to a record in the stream. For example, if all records in the stream were flushed, *plsnLastFlushed* is the LSN that will be assigned to the next record written to the stream.

Return Value

ClfsFlushToLsn returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsFlushBuffers

ClfsGetContainerName function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsGetContainerName** routine returns the path name of a specified container.

Syntax

```
CLFSUSER_API NTSTATUS ClfsGetContainerName(
  PLOG_FILE_OBJECT  plfoLog,
  CLFS_CONTAINER_ID cidLogicalContainer,
  PUNICODE_STRING   puszContainerName,
  PULONG            pcActualLenContainerName
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS log. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`cidLogicalContainer`

The logical identifier of a container in the CLFS log.

`puszContainerName`

A pointer to a UNICODE_STRING structure. The **Buffer** member of *puszContainerName* must point to a caller-allocated buffer that receives the path name of the container. The **MaximumLength** member must supply the size of the buffer.

`pcActualLenContainerName`

A pointer to a ULONG-typed variable that receives the actual length, in Unicode characters, of the container's path name. This parameter can be **NULL**.

Return Value

ClfsGetContainerName returns one of the status codes defined in Ntstatus.h. The following table gives the meanings of various return values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The **Buffer** member of *puszContainerName* receives the full path name of the container. If *pcActualLenContainerName* is not **NULL**, it receives the actual length, in Unicode characters, of the full path name.

STATUS_BUFFER_OVERFLOW	The actual length, in Unicode characters, of the full path name is larger than *puszContainerName*->**MaximumLength**. The **Buffer** member of *puszContainerName* receives the full path name of the container truncated to the number of characters specified by *puszContainerName*->**MaximumLength**. If *pcActualLenContainerName* is not **NULL**, it receives the actual length, in Unicode characters, of the full path name.
Other	On return, the contents of *puszContainerName*->**Buffer** and *pcActualLenContainerName* are undefined.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainer

ClfsRemoveLogContainer

ClfsGetIoStatistics function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsGetIoStatistics** routine returns I/O statistics for a specified CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsGetIoStatistics(
  PLOG_FILE_OBJECT   plfoLog,
  PVOID              pvStatsBuffer,
  ULONG              cbStatsBuffer,
  CLFS_IOSTATS_CLASS eStatsClass,
  PULONG             pcbStatsWritten
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. The I/O statistics will be collected for the stream's underlying log on stable storage. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`pvStatsBuffer`

A pointer to a CLFS_IO_STATISTICS structure that receives the I/O statistics data.

`cbStatsBuffer`

The size, in bytes, of the buffer pointed to by *pvStatsBuffer*. This parameter must be at least **sizeof**(CLFS_IO_STATISTICS_HEADER). If this parameter is less than **sizeof**(**CLFS_IO_STATISTICS**), *pvStatsBuffer* will receive only a portion of the available I/O statistics.

`eStatsClass`

This parameter is reserved for future use.

`pcbStatsWritten`

A pointer to a ULONG-typed variable that receives the number of bytes actually written to *pvStatsBuffer*. This parameter can be **NULL**.

Return Value

ClfsGetIoStatistics returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCreateLogFile

ClfsLsnBlockOffset function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnBlockOffset** routine returns the sector-aligned block offset contained in a specified LSN.

Syntax

```
CLFSUSER_API ULONG ClfsLsnBlockOffset(
  const CLFS_LSN *plsn
);
```

Parameters

`plsn`

A pointer to a CLFS_LSN structure from which the block offset is retrieved.

Return Value

ClfsLsnBlockOffset returns the block offset contained in the LSN that is supplied by the caller.

Remarks

The block offset returned by this routine is a multiple of the sector size on the stable storage medium. For example, if the sector size is 1024 bytes, the block offset is a multiple of 1024.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnContainer

ClfsLsnCreate

ClfsLsnRecordSequence

ClfsLsnContainer function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnContainer** routine returns the logical container identifier contained in a specified LSN.

Syntax

```
CLFSUSER_API CLFS_CONTAINER_ID ClfsLsnContainer(
  const CLFS_LSN *plsn
);
```

Parameters

`plsn`

A pointer to a CLFS_LSN structure from which the container identifier is retrieved.

Return Value

ClfsLsnContainer returns the logical container identifier contained in the LSN that is supplied by the caller. The logical container identifier is not necessarily the same as the identifier of the physical container on stable storage.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnBlockOffset

ClfsLsnCreate

ClfsLsnRecordSequence

ClfsLsnCreate function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnCreate** routine creates a log sequence number (LSN), given a container identifier, a block offset, and a record sequence number.

Syntax

```
CLFSUSER_API CLFS_LSN ClfsLsnCreate(
  CLFS_CONTAINER_ID cidContainer,
  ULONG             offBlock,
  ULONG             cRecord
);
```

Parameters

`cidContainer`

An integer in the range 0x0 through 0xFFFFFFFF that supplies the container identifier.

`offBlock`

The block offset. This parameter must be a multiple of 512.

`cRecord`

An integer in the range 0 - 511 that supplies the record sequence number.

Return Value

ClfsLsnCreate returns a CLFS_LSN structure that represents the container identifier, block offset, and record sequence number supplied by the caller.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

systems.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnBlockOffset

ClfsLsnContainer

ClfsLsnRecordSequence

ClfsLsnEqual function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnEqual** routine determines whether two LSNs from the same stream are equal.

Syntax

```
CLFSUSER_API BOOLEAN ClfsLsnEqual(
  const CLFS_LSN *plsn1,
  const CLFS_LSN *plsn2
);
```

Parameters

`plsn1`

A pointer to a CLFS_LSN structure that supplies one of the LSNs to be compared.

`plsn2`

A pointer to a CLFS_LSN structure that supplies the other LSN to be compared.

Return Value

ClfsLsnEqual returns **TRUE** if the two LSNs are equal; otherwise, it returns **FALSE**.

Remarks

CLFS_LSN_NULL (the smallest LSN) and CLFS_LSN_INVALID (larger than any valid LSN) are valid arguments to **ClfsLsnEqual**.

LSNs from different streams are not comparable. Do not use **ClfsLsnEqual**, ClfsLsnGreater, and the like to compare LSNs from different streams.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys

IRQL	Any level

See Also

ClfsLsnGreater

ClfsLsnLess

ClfsLsnNull

ClfsLsnGreater function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnGreater** routine determines whether one LSN is greater than another LSN. The two LSNs must be from the same stream.

Syntax

```
CLFSUSER_API BOOLEAN ClfsLsnGreater(
  const CLFS_LSN *plsn1,
  const CLFS_LSN *plsn2
);
```

Parameters

`plsn1`

A pointer to a CLFS_LSN structure that supplies one of the LSNs to be compared.

`plsn2`

A pointer to a CLFS_LSN structure that supplies the other LSN to be compared.

Return Value

ClfsLsnGreater returns **TRUE** if *plsn1* is strictly greater than *plsn2*; otherwise, it returns **FALSE**.

Remarks

CLFS_LSN_NULL (the smallest LSN) and CLFS_LSN_INVALID (larger than any valid LSN) are valid arguments to **ClfsLsnGreater**.

LSNs from different streams are not comparable. Do not use ClfsLsnEqual, **ClfsLsnGreater**, and the like to compare LSNs from different streams.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnEqual

ClfsLsnLess

ClfsLsnNull

ClfsLsnLess function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnLess** routine determines whether one LSN is less than another LSN. The two LSNs must be from the same stream.

Syntax

```
CLFSUSER_API BOOLEAN ClfsLsnLess(
  const CLFS_LSN *plsn1,
  const CLFS_LSN *plsn2
);
```

Parameters

`plsn1`

A pointer to a CLFS_LSN structure that supplies one of the LSNs to be compared.

`plsn2`

A pointer to a CLFS_LSN structure that supplies the other LSN to be compared.

Return Value

ClfsLsnLess returns **TRUE** if *plsn1* is strictly less than *plsn2*; otherwise, it returns **FALSE**.

Remarks

CLFS_LSN_NULL (the smallest LSN) and CLFS_LSN_INVALID (larger than any valid LSN) are valid arguments to **ClfsLsnLess**.

LSNs from different streams are not comparable. Do not use ClfsLsnEqual, **ClfsLsnLess** and the like to compare LSNs from different streams.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib

DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnEqual

ClfsLsnGreater

ClfsLsnNull

ClfsLsnNull function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnNull** routine determines whether a specified LSN is equal to the smallest possible LSN, CLFS_LSN_NULL.

Syntax

```
CLFSUSER_API BOOLEAN ClfsLsnNull(
  const CLFS_LSN *plsn
);
```

Parameters

`plsn`

A pointer to a CLFS_LSN structure that supplies the LSN to be tested.

Return Value

ClfsLsnNull returns **TRUE** if the LSN supplied by the caller is equal to CLFS_LSN_NULL; otherwise, it returns **FALSE**.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnEqual

ClfsLsnGreater

ClfsLsnLess

ClfsLsnRecordSequence function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsLsnRecordSequence** routine returns the record sequence number contained in a specified LSN.

Syntax

```
CLFSUSER_API ULONG ClfsLsnRecordSequence(
  const CLFS_LSN *plsn
);
```

Parameters

`plsn`

A pointer to a CLFS_LSN structure from which the record sequence number is retrieved.

Return Value

ClfsLsnRecordSequence returns the record sequence number contained in the LSN that is supplied by the caller.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	Any level

See Also

ClfsLsnBlockOffset

ClfsLsnContainer

ClfsMgmtDeregisterManagedClient function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtDeregisterManagedClient** routine removes the connection between a client and a log, so that the client no longer manages the log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtDeregisterManagedClient(
  CLFS_MGMT_CLIENT ClientCookie
);
```

Parameters

`ClientCookie`

The cookie we give back to the caller for future use of the API.

Return Value

The **ClfsMgmtDeregisterManagedClient** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The client is no longer associated with the log.
STATUS_UNSUCCESSFUL	CLFS management was not able to deregister the client.
STATUS_INVALID_PARAMETER	A **NULL** value was supplied for the *Client* parameter.

STA **TUS** **_IN** **VAL** **ID_** **PAR** **AM** **ETE** **R_1**	The value that was supplied for the *Client* parameter does not represent a valid client.

Remarks

After calling the **ClfsMgmtDeregisterManagedClient** routine, a client's callback functions will no longer be invoked.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsMgmtRegisterManagedClient

ClfsMgmtHandleLogFileFull function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtHandleLogFileFull** routine attempts to make more space available in a log. It might make more space available by adding containers to the log, or it might ask clients to move their log tails.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtHandleLogFileFull(
  CLFS_MGMT_CLIENT Client
);
```

Parameters

`Client`

The client that is requesting CLFS management to make space available in the log. The value of this parameter should be the **CLFS_MGMT_CLIENT** structure that is obtained through a call to the ClfsMgmtRegisterManagedClient routine.

Return Value

The **ClfsMgmtHandleLogFileFull** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The client is not managing a log.
STATUS_INVALID_PARAMETER_1	The value that was supplied for the *Client* parameter either was **NULL** or does not represent a valid client.

STATUS_LOG_FULL_HANDLER_IN_PROGRESS	CLFS management is already attempting to resolve a log file full condition for this client.
STATUS_UNSUCCESSFUL	CLFS management was not able to process the request.
STATUS_PENDING	CLFS management is processing the request to create space in the log.
STATUS_SUCCESS	The requested amount of space is available in the log.

Remarks

To make more space available in the log, the **ClfsMgmtHandleLogFileFull** routine first tries to add more containers to the log in accordance with the growth rate, new container size, and maximum size policies. If more containers cannot be added, then the **ClfsMgmtHandleLogFileFull** routine tries to free existing space by invoking one or more clients' ClfsAdvanceTailCallback functions.

If the **ClfsMgmtHandleLogFileFull** routine returns STATUS_PENDING, the client's ClfsLogGrowthCompleteCallback function will be called when the request to make space in the log has been completed.

If the **ClfsMgmtHandleLogFileFull** routine returns STATUS_LOG_FULL_HANDLER_IN_PROGRESS, the client has already requested CLFS management to handle a log file full condition. The client's ClfsLogGrowthCompleteCallback function will not be called a second time.

If the **ClfsMgmtHandleLogFileFull** routine returns STATUS_SUCCESS, the call completed synchronously, and the client's ClfsLogGrowthCompleteCallback function will not be invoked.

If the **ClfsMgmtHandleLogFileFull** routine returns STATUS_PENDING, then CLFS management is in the process of trying to free space in the log, and will call the client's ClfsLogGrowthCompleteCallback function when the log file full condition has been handled. If the log is pinned, CLFS management will call the client's

ClfsLogGrowthCompleteCallback function with the *LogIsPinned* parameter set to **TRUE** before the **ClfsMgmtHandleLogFileFull** routine returns STATUS_PENDING.

> **Important** It is possible that the client's ClfsLogGrowthCompleteCallback function could be called before the call to **ClfsMgmtHandleLogFileFull** returns.

> **Important** If the **ClfsMgmtHandleLogFileFull** routine returns STATUS_PENDING, you should not call the **ClfsMgmtHandleLogFileFull** routine again for this client until the client's ClfsLogGrowthCompleteCallback function has been called.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAdvanceTailCallback

ClfsLogGrowthCompleteCallback

ClfsMgmtRegisterManagedClient

ClfsMgmtInstallPolicy function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtInstallPolicy** routine adds a CLFS_MGMT_POLICY structure to a physical log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtInstallPolicy(
  PLOG_FILE_OBJECT  LogFile,
  PCLFS_MGMT_POLICY Policy,
  ULONG             PolicyLength
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log that this instance of the **CLFS_MGMT_POLICY** structure will apply to. The policy applies to all streams within the log, even if a single stream within the log was specified.

`Policy`

A pointer to a CLFS_MGMT_POLICY structure that contains the policy to be installed.

`PolicyLength`

The length, in bytes, of the structure pointed to by the *Policy* parameter.

Return Value

The **ClfsMgmtInstallPolicy** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The policy has been installed.
STA TUS _IN VAL ID_ PAR AM ETE R_1	A **NULL** value was supplied for the *LogFile* parameter.

STATUS_INVALID_PARAMETER_2	A **NULL** value was supplied for the *Policy* parameter.
STATUS_INVALID_PARAMETER_3	The value of the *PolicyLength* parameter is less than the size of an instance of the CLFS_MGMT_POLICY structure.
STATUS_NOT_SUPPORTED	The **PolicyFlags** member of the **CLFS_MGMT_POLICY_STRUCTURE** pointed to by the *Policy* parameter specifies any flag other than LOG_POLICY_OVERWRITE.
STATUS_INVALID_PARAMETER	One of the following conditions is true: • The *Policy* parameter's **Version** member is not equal to CLFS_MGMT_POLICY_VERSION. • The value of the *Policy*parameter is equal to **ClfsMgmtPolicyInvalid**. • The *Policy* parameter's **PolicyFlags** member specifies any flag other than LOG_POLICY_OVERWRITE.
STATUS_LOG_POLICY_ALREADY_INSTALLED	The log already has a policy of this type, and the LOG_POLICY_OVERWRITE flag is not set.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	There is insufficient memory to complete the operation.

This routine might also return other NTSTATUS Values.

Remarks

Policies are volatile. When all handles to the log are closed, the policies will be lost. You should install policies each time you register the first client.

You should only register a **CLFS_MGMT_POLICY** structure whose **PolicyType** member is equal to **ClfsMgmtPolicyNewContainerSize** before the first container in the log is created. Any subsequent registrations are ignored.

The log policy that is installed applies to the physical log, even if the *LogFile* parameter specifies a log stream.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_MGMT_POLICY

ClfsMgmtQueryPolicy function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtQueryPolicy** routine retrieves a specific CLFS_MGMT_POLICY structure for a log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtQueryPolicy(
  PLOG_FILE_OBJECT      LogFile,
  CLFS_MGMT_POLICY_TYPE PolicyType,
  PCLFS_MGMT_POLICY     Policy,
  PULONG                PolicyLength
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log whose policy is being retrieved.

`PolicyType`

A value of the CLFS_MGMT_POLICY_TYPE enumeration that identifies the type of policy to be retrieved.

`Policy`

An instance of the CLFS_MGMT_POLICY structure that contains the policy.

`PolicyLength`

The length of the *Policy* parameter.

Return Value

The **ClfsMgmtQueryPolicy** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	CLFS management has retrieved the requested policy.
STATUS_UNSUCCESSFUL	CLFS management was not able to process the request.

STATUS_INVALID_PARAMETER	The value of the *PolicyType* parameter is not valid for the **CLFS_MGMT_POLICY_TYPE** enumeration.
STATUS_INVALID_PARAMETER_1	A **NULL** value was supplied for the *LogFile* parameter.
STATUS_INVALID_PARAMETER_2	The value of the *PolicyLength* parameter is less than the size of an instance of the **CLFS_MGMT_POLICY** structure.
STATUS_INSUFFICIENT_RESOURCES	There is insufficient memory to complete the operation.
STATUS_LOG_POLICY_NOT_INSTALLED	No policy of this type has been registered for the log file.

This routine might also return other NTSTATUS Values.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_MGMT_POLICY

CLFS_MGMT_POLICY_TYPE

ClfsMgmtRegisterManagedClient function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtRegisterManagedClient** routine creates a client that will manage a CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtRegisterManagedClient(
  PLOG_FILE_OBJECT               LogFile,
  PCLFS_MGMT_CLIENT_REGISTRATION RegistrationData,
  PCLFS_MGMT_CLIENT              ClientCookie
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log stream that the client created by **ClfsMgmtRegisterManagedClient** will manage.

`RegistrationData`

An instance of the CLFS_MGMT_CLIENT_REGISTRATION structure that contains the functions that will be used to manage the log.

`ClientCookie`

The cookie we give back to the caller for future use of the API.

Return Value

The **ClfsMgmtRegisterManagedClient** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	A client has been created to manage the log.
STA TUS _UN SUC CES SFU L	CLFS management was not able to create a client to manage the log.

STA TUS _IN SUF FICI ENT _RE SO URC ES	There is insufficient memory to complete the operation.
STA TUS _IN VAL ID_ PAR AM ETE R	An input parameter is invalid.

This routine might also return other NTSTATUS Values.

Remarks

The *Client* parameter that is provided by the **ClfsMgmtRegisterManagedClient** routine is a required parameter for other CLFS management functions. You should store this value for later use.

A client application must be ready for its callback routine to be called as soon as it calls the **ClfsMgmtRegisterManagedClient** routine.

More than one client can register with a log stream.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_MGMT_CLIENT_REGISTRATION

ClfsMgmtRemovePolicy function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtRemovePolicy** routine resets a log's CLFS_MGMT_POLICY structure to its default value.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtRemovePolicy(
  PLOG_FILE_OBJECT      LogFile,
  CLFS_MGMT_POLICY_TYPE PolicyType
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log whose policy is being removed.

`PolicyType`

A value of the CLFS_MGMT_POLICY_TYPE enumeration that supplies the type of the policy to be removed.

Return Value

The **ClfsMgmtRemovePolicy** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	CLFS management has removed the requested policy.
STATUS_UNSUCCESSFUL	CLFS management was not able to process the request.

STATUS_INVALID_PARAMETER_1	A **NULL** value was supplied for the *LogFile* parameter.
STATUS_INSUFFICIENT_RESOURCES	There is insufficient memory to complete the operation.
STATUS_INVALID_PARAMETER	The value of the *PolicyType* parameter is not a valid value for the **CLFS_MGMT_POLICY_TYPE** enumeration.
STATUS_LOG_POLICY_NOT_INSTALLED	No policy of this type has been registered for the log file.

This routine might also return other NTSTATUS Values.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_MGMT_POLICY

CLFS_MGMT_POLICY_TYPE

ClfsMgmtSetLogFileSize function

1/25/2019 • 3 minutes to read • Edit Online

The **ClfsMgmtSetLogFileSize** routine adds containers to a log or deletes containers from a log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtSetLogFileSize(
  PLOG_FILE_OBJECT                    LogFile,
  PULONGLONG                          NewSizeInContainers,
  PULONGLONG                          ResultingSizeInContainers,
  PCLFS_SET_LOG_SIZE_COMPLETE_CALLBACK CompletionRoutine,
  PVOID                               CompletionRoutineData
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log, or a stream within the log, to which containers are being added or deleted.

`NewSizeInContainers`

A pointer to the requested log size. The caller sets this parameter to one of the following values.

VALUE	MEANING
0	Enforce the minimum size policy. For more information about this policy, see ClfsMgmtInstallPolicy. If a minimum size policy is not installed, one of the following occurs: • If the log currently has less than 2 containers, the log will be expanded to a size of 2 containers. • If the log currently has 2 or more containers, no changes are made and the call succeeds. If a minimum size policy is installed, one of the following occurs: • If the log currently has less than the minimum number of containers specified by the minimum size policy, the log expands to the policy-specified minimum number of containers. • If the number of containers in the log is greater than or equal to the minimum number of containers specified by the minimum size policy, no changes are made and the call succeeds with no error.
1	Invalid value. The call fails and returns STATUS_INVALID_VALUE.

2 to 1023	The desired size of the log, expressed as the number of containers. If this number is smaller than the minimum number of containers specified by the installed policy, the call fails with ERROR_COULD_NOT_RESIZE_LOG. If this number is larger than the maximum number of containers specified by the installed policy, the log expands only as far as the policy-specified maximum number of containers, and the call succeeds with no error.
1024 to MAXULONGLONG	If no maximum size policy is installed, the call fails and returns ERROR_LOG_POLICY_CONFLICT. If a maximum size policy is installed, the log expands to the maximum number of containers specified by the maximum size policy and the call succeeds with no error.

To determine the actual log size, which might be different from the requested size, use the *ResultingSizeInContainers* parameter.

`ResultingSizeInContainers`

A pointer to the resulting log size. If successful, the routine writes the actual size of the log, expressed as the number of containers in the log, to the location pointed to by this parameter.

`CompletionRoutine`

Not used. Set to NULL.

`CompletionRoutineData`

Not used. Set to NULL.

Return Value

The **ClfsMgmtSetLogFileSize** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The log file size has been set. The *ResultingSizeInContainers* parameter contains the current size of the log.
STATUS_UNSUCCESSFUL	CLFS management was not able to set the log file size.

STATUS_INVALID_PARAMETER_1	The value of the *LogFile* parameter is **NULL**, or the contents of the *NewSizeInContainers* parameter is 1.
STATUS_INVALID_PARAMETER_2	The value of the *NewSizeInContainers* parameter is **NULL**.
STATUS_LOG_POLICY_INVALID	There is a conflict between the maximum size and minimum size policies for the log.
STATUS_COULD_NOT_RESIZE_LOG	CLFS management could not delete enough containers to reach *NewSizeInContainers*.
STATUS_LOG_POLICY_CONFLICT	CLFS management could not add enough containers to the log to reach *NewSizeInContainers*. This might be due to a conflict with a policy that the client set.

This routine might also return other NTSTATUS Values.

Remarks

The **ClfsMgmtSetLogFileSize** routine is typically used only when a client starts or stops. Do not call the **ClfsMgmtSetLogFileSize** routine from within your ClfsAdvanceTailCallback function.

Requirements

Minimum supported client	Available starting with Windows Server 2003 R2 and Windows Vista.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_MGMT_POLICY_TYPE

ClfsMgmtInstallPolicy

ClfsMgmtSetLogFileSizeAsClient function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtSetLogFileSizeAsClient** routine sets the log file size by adding containers to a client log or deleting containers from a client log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtSetLogFileSizeAsClient(
  PLOG_FILE_OBJECT                    LogFile,
  PCLFS_MGMT_CLIENT                   ClientCookie,
  PULONGLONG                          NewSizeInContainers,
  PULONGLONG                          ResultingSizeInContainers,
  PCLFS_SET_LOG_SIZE_COMPLETE_CALLBACK CompletionRoutine,
  PVOID                               CompletionRoutineData
);
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the Common Log File System (CLFS) log, or a stream within the log, to which containers are being added or deleted. The value of this parameter is obtained through a call to the ClfsCreateLogFile routine.

`ClientCookie`

A pointer to a client-supplied cookie. The value of this parameter should be the **CLFS_MGMT_CLIENT** structure that is obtained through a call to the ClfsMgmtRegisterManagedClient routine.

`NewSizeInContainers`

The desired size of the log, expressed in the number of containers. There can be at most 1,024 containers for a log file.

`ResultingSizeInContainers`

The actual size of the log, expressed in the number of containers.

`CompletionRoutine`

Not used.

`CompletionRoutineData`

Not used.

Return Value

The **ClfsMgmtSetLogFileSizeAsClient** routine returns an NTSTATUS value.

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The log file size has been set.
STATUS_INVALID_PARAMETER_1	The value of the *LogFile* parameter is **NULL**, or the contents of the *NewSizeInContainers* parameter is 1.
STATUS_INVALID_PARAMETER_2	The value of the *NewSizeInContainers* parameter is **NULL**.
STATUS_LOG_POLICY_INVALID	The installed set of policies on the log is invalid. This might be due to an invalid ClfsMgmtPolicyAutoShrink policy or ClfsMgmtPolicyMaximumSize policy.
STATUS_COULD_NOT_RESIZE_LOG	CLFS management could not delete enough containers to reach the value in *NewSizeInContainers*.

STA TUS _LO G_P OLI CY_ CO NFL ICT	A policy on the specified log prevented the operation from completing. This may occur if CLFS management could not add enough containers to the log to reach the value in *NewSizeInContainers*. This might be due to a conflict with a policy that the client set.

This routine might also return other NTSTATUS values.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Desktop
Header	wdm.h
DLL	Clfs.sys

See Also

CLFS Management Library Routines

ClfsMgmtTailAdvanceFailure function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsMgmtTailAdvanceFailure** routine notifies CLFS management that the client could not advance the log's tail.

Syntax

```
CLFSUSER_API NTSTATUS ClfsMgmtTailAdvanceFailure(
  CLFS_MGMT_CLIENT Client,
  NTSTATUS         Reason
);
```

Parameters

`Client`

A pointer to the client. This is the value that was obtained through a call to the ClfsMgmtRegisterManagedClient routine.

`Reason`

A value that indicates why the log's tail could not be advanced.

Return Value

The **ClfsMgmtTailAdvanceFailure** routine returns one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	CLFS management has processed the notification that the log's tail could not be advanced.
STATUS_INVALID_PARAMETER	A **NULL** value was supplied for the *Client* parameter.

STA **TUS** **_IN** **VAL** **ID_** **PAR** **AM** **ETE** **R_1**	The value that was supplied for the *Client* parameter does not represent a valid client.

Remarks

If a client cannot advance its log's tail to or beyond the requested LSN, then the client must call the **ClfsMgmtTailAdvanceFailure** routine to indicate that it is not able to advance its tail. Until the client either moves its tail as requested or calls the **ClfsMgmtTailAdvanceFailure** routine, the client will not receive any further requests to move its tail.

The value of the *Reason* parameter is passed back to the client as the value of the *OperationStatus* parameter when the ClfsLogGrowthCompleteCallback function is invoked.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsMgmtRegisterManagedClient

ClfsQueryLogFileInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsQueryLogFileInformation** routine returns metadata and state information for a specified CLFS stream or its underlying physical log or both.

Syntax

```
CLFSUSER_API NTSTATUS ClfsQueryLogFileInformation(
  PLOG_FILE_OBJECT           plfoLog,
  CLFS_LOG_INFORMATION_CLASS eInformationClass,
  PVOID                      pinfoInputBuffer,
  ULONG                      cbinfoInputBuffer,
  PVOID                      pinfoBuffer,
  PULONG                     pcbInfoBuffer
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`eInformationClass`

A value from the CLFS_LOG_INFORMATION_CLASS enumeration that specifies the type of information being requested.

`pinfoInputBuffer`

Optional input buffer which is required for some information classes.

`cbinfoInputBuffer`

Length in bytes of pinfoInputBuffer.

`pinfoBuffer`

A pointer to a buffer that receives the log information. The structure of this buffer depends on the value of *eInformationClass*. The following table shows the relationship between the information class and the data type of the buffer.

VALUE OF *EINFORMATIONCLASS*	TYPE OF BUFFER POINTED TO BY *PINFOBUFFER*
ClfsLogBasicInformation	CLFS_INFORMATION
ClfsLogBasicInformationPhysical	CLFS_INFORMATION
ClfsLogNameInformation	CLFS_LOG_NAME_INFORMATION
ClfsLogPhysicalNameInformation	CLFS_LOG_NAME_INFORMATION

ClfsLogStreamIdentifierInformation	CLFS_STREAM_ID_INFORMATION

`pcbInfoBuffer`

Length in bytes of the incoming information buffer.

Return Value

ClfsQueryLogFileInformation returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsSetLogFileInformation

ClfsReadLogRecord function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsReadLogRecord** routine reads a target record in a CLFS stream and returns a read context that the caller can use to read the records preceding or following it in the stream.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReadLogRecord(
  PVOID             pvMarshalContext,
  PCLFS_LSN         plsnFirst,
  CLFS_CONTEXT_MODE peContextMode,
  PVOID             *ppvReadBuffer,
  PULONG            pcbReadBuffer,
  PCLFS_RECORD_TYPE peRecordType,
  PCLFS_LSN         plsnUndoNext,
  PCLFS_LSN         plsnPrevious,
  PVOID             *ppvReadContext
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`plsnFirst`

A pointer to a CLFS_LSN structure that supplies the LSN of the target record in the log.

`peContextMode`

A value from the CLFS_CONTEXT_MODE enumeration that specifies the initial mode (ClfsContextUndoNext, ClfsContextPrevious, or ClfsContextForward) of the read context returned in *ppvReadContext*.

`ppvReadBuffer`

A pointer to a variable that receives a pointer to the data buffer of the target record.

`pcbReadBuffer`

A pointer to a ULONG-typed variable that receives the size, in bytes, of the buffer pointed to by **ppvReadBuffer*. This is the length of the data buffer of the target record.

`peRecordType`

A pointer to a variable of type CLFS_RECORD_TYPE that receives one of the following values.

VALUE	MEANING
ClfsDataRecord	The target record is a data record.
ClfsRestartRecord	The target record is a restart record.

`plsnUndoNext`

A pointer to a **CLFS_LSN** structure that receives the undo-next LSN of the target record.

`plsnPrevious`

A pointer to a **CLFS_LSN** structure that receives the previous LSN of the target record.

`ppvReadContext`

A pointer to a variable that receives a pointer to an opaque read context. The client repeatedly passes this context to ClfsReadNextLogRecord to read the remaining records in a sequence. After the client has read all the records in the sequence, it calls ClfsTerminateReadLog to free the read context.

Return Value

ClfsReadLogRecord returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

You must call **ClfsTerminateReadLog** when you have finished using the read context in order to avoid a memory leak.

For an explanation of CLFS concepts and terminology, see Common Log File System.

For information about reading records from CLFS streams, see Reading Data Records from a CLFS Stream and Reading Restart Records from a CLFS Stream.

Read contexts are not thread-safe. Clients are responsible for serializing access to read contexts.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsReadNextLogRecord

ClfsTerminateReadLog

ClfsReadNextLogRecord function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsReadNextLogRecord** routine reads the next record in a sequence, relative to the current record in a read context.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReadNextLogRecord(
  PVOID             pvReadContext,
  PVOID             *ppvBuffer,
  PULONG            pcbBuffer,
  PCLFS_RECORD_TYPE peRecordType,
  PCLFS_LSN         plsnUser,
  PCLFS_LSN         plsnUndoNext,
  PCLFS_LSN         plsnPrevious,
  PCLFS_LSN         plsnRecord
);
```

Parameters

`pvReadContext`

A pointer to a read context that the caller previously obtained by calling ClfsReadLogRecord or ClfsReadRestartArea.

`ppvBuffer`

A pointer to a variable that receives a pointer to a buffer that contains the record data.

`pcbBuffer`

A pointer to a ULONG-typed variable that receives the size, in bytes, of the buffer pointed to by **ppvBuffer*. This is the length of the data buffer of the record read.

`peRecordType`

A pointer to a variable of type CLFS_RECORD_TYPE. The caller must set this parameter to one of the following values.

VALUE	RECORD THAT WILL BE READ
ClfsDataRecord	The next data record.
ClfsRestartRecord	The next restart record.
ClfsClientRecord	The next record that is either a data record or a restart record.

`plsnUser`

A pointer to a CLFS_LSN structure that supplies the LSN of the record to be read. The specified record is read instead of the record that would have been read according to the mode (**ClfsContextUndoNext**, **ClfsContextPrevious**, or **ClfsContextForward**) of the read context (*pvReadContext*). The LSN supplied in

plsnUser must be less than the current LSN of the read context. This parameter can be **NULL**.

`plsnUndoNext`

A pointer to a **CLFS_LSN** structure that receives the undo-next LSN of the record that is read.

`plsnPrevious`

A pointer to a **CLFS_LSN** structure that receives the previous LSN of the record that was read.

`plsnRecord`

A pointer to a **CLFS_LSN** structure that receives the LSN of the record that was read.

Return Value

ClfsReadNextLogRecord returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

For information about reading records from CLFS streams, see Reading Data Records from a CLFS Stream and Reading Restart Records from a CLFS Stream.

Read contexts are not thread-safe. Clients are responsible for serializing access to read contexts.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

ClfsReadLogRecord

ClfsReadRestartArea

ClfsReadPreviousRestartArea function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsReadPreviousRestartArea** routine reads the previous restart record relative to the current record in a read context.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReadPreviousRestartArea(
  PVOID     pvReadContext,
  PVOID     *ppvRestartBuffer,
  PULONG    pcbRestartBuffer,
  PCLFS_LSN plsnRestart
);
```

Parameters

`pvReadContext`

A pointer to a read context obtained by a previous call to ClfsReadRestartArea.

`ppvRestartBuffer`

A pointer to a variable that receives a pointer to the data buffer of the restart record that is read.

`pcbRestartBuffer`

A pointer to a ULONG-typed variable that receives the size, in bytes, of the buffer pointed to by **ppvRestartBuffer*. This is the length of the data buffer of the restart record that is read.

`plsnRestart`

A pointer to a CLFS_LSN structure that receives the previous LSN of the restart record that was read.

Return Value

ClfsReadPreviousRestartArea returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h. If there are no previous restart areas, the return value is STATUS_LOG_START_OF_LOG.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

For information about reading records from CLFS streams, see Reading Data Records from a CLFS Stream and Reading Restart Records from a CLFS Stream.

Read contexts are not thread-safe. Clients are responsible for serializing access to read contexts.

Clients append restart records to a CLFS log by calling ClfsWriteRestartArea.

With the **ReadPreviousLogRestartArea** routine, clients can implement a form of fuzzy checkpointing. For example, the checkpoint state could consist of the log records delimited by two or more restart records.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

ClfsReadRestartArea

ClfsWriteRestartArea

ClfsReadRestartArea function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsReadRestartArea** routine reads the restart record that was most recently written to a specified CLFS stream.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReadRestartArea(
  PVOID     pvMarshalContext,
  PVOID     *ppvRestartBuffer,
  PULONG    pcbRestartBuffer,
  PCLFS_LSN plsn,
  PVOID     *ppvReadContext
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`ppvRestartBuffer`

A pointer to a variable that receives a pointer to the data buffer of the restart record.

`pcbRestartBuffer`

A pointer to a ULONG-typed variable that receives the size, in bytes, of the data buffer pointed to by *pcbRestartBuffer*. This is the length of the data buffer of the restart record.

`plsn`

A pointer to a CLFS_LSN structure that receives the LSN of the restart record that was read.

`ppvReadContext`

A pointer to a variable that receives a pointer to an opaque read context. The caller can pass this context to ClfsReadPreviousRestartArea or ClfsReadNextLogRecord. When the caller has finished using the read context, it must free the context by calling ClfsTerminateReadLog.

Return Value

ClfsReadRestartArea returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

For information about reading records from CLFS streams, see Reading Data Records from a CLFS Stream and Reading Restart Records from a CLFS Stream.

Read contexts are not thread-safe. Clients are responsible for serializing access to read contexts.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsCreateMarshallingArea

ClfsReadNextLogRecord

ClfsReadPreviousRestartArea

ClfsTerminateReadLog

ClfsRemoveLogContainer function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsRemoveLogContainer** routine removes a container from a CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsRemoveLogContainer(
  PLOG_FILE_OBJECT plfoLog,
  PUNICODE_STRING  puszContainerPath,
  BOOLEAN          fForce
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS log from which the container will be removed. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`puszContainerPath`

A pointer to a UNICODE_STRING structure that supplies the path name for the container to be removed. The path name was created in a previous call to ClfsAddLogContainer or ClfsAddLogContainerSet. The path can be absolute or relative to the location of the base log file for the CLFS log that is represented by *plfoLog*. Paths that are relative to the base log file must begin with CLFS_CONTAINER_RELATIVE_PREFIX, which is the string literal (L"%BLF%\"). The directories "." and ".." are not allowed in a relative path.

`fForce`

A Boolean value that specifies whether the container removal is forced (**TRUE**) or lazy (**FALSE**).

Return Value

ClfsRemoveLogContainer returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

Forced container removal (*fForce* = **TRUE**) succeeds only if the container to be removed is not part of the active log.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.

Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainer

ClfsAddLogContainerSet

ClfsCreateLogFile

ClfsRemoveLogContainerSet

LOG_FILE_OBJECT

UNICODE_STRING

ClfsRemoveLogContainerSet function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsRemoveLogContainerSet** routine atomically removes a set of containers from a CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsRemoveLogContainerSet(
  PLOG_FILE_OBJECT plfoLog,
  USHORT           cContainers,
  PUNICODE_STRING  rgwszContainerPath,
  BOOLEAN          fForce
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log from which the containers will be removed. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`cContainers`

The number of containers in the set. This is the number of elements in the *rgwszContainerPath* array.

`rgwszContainerPath`

A pointer to an array of UNICODE_STRING structures, each of which supplies the path name for one of the containers to be removed. The number of elements in the array is given by *cContainers*. A given path can be absolute or relative to the location of the base log file for the CLFS log represented by *plfoLog*. Paths that are relative to the base log file must begin with CLFS_CONTAINER_RELATIVE_PREFIX, which is the string literal (L"%BLF%\"). The directories "." and ".." are not allowed in a relative path.

`fForce`

A Boolean value that specifies whether the container removal is forced (**TRUE**) or lazy (**FALSE**).

Return Value

ClfsRemoveLogContainerSet returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

Forced container removal (*fForce* = **TRUE**) succeeds only if the containers to be removed are not part of the active log.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

ClfsAddLogContainerSet

ClfsCreateLogFile

ClfsRemoveLogContainer

LOG_FILE_OBJECT

UNICODE_STRING

ClfsReserveAndAppendLog function

1/25/2019 • 3 minutes to read • Edit Online

The **ClfsReserveAndAppendLog** routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReserveAndAppendLog(
  PVOID             pvMarshalContext,
  PCLFS_WRITE_ENTRY rgWriteEntries,
  ULONG             cWriteEntries,
  PCLFS_LSN         plsnUndoNext,
  PCLFS_LSN         plsnPrevious,
  ULONG             cReserveRecords,
  PLONGLONG         rgcbReservation,
  ULONG             fFlags,
  PCLFS_LSN         plsn
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`rgWriteEntries`

A pointer to an array of CLFS_WRITE_ENTRY structures, each of which holds a pointer to a buffer of data that will become part of the record that is appended to the log. This parameter can be **NULL** if *cWriteEntries* is zero.

`cWriteEntries`

The number of elements in the array pointed to by *rgWriteEntries*. This parameter must be zero if *rgWriteEntries* is **NULL**.

`plsnUndoNext`

A pointer to a CLFS_LSN structure that supplies the undo-next LSN of the record to be appended.

`plsnPrevious`

A pointer to a CLFS_LSN structure that supplies the previous LSN of the record to be appended.

`cReserveRecords`

The number of elements in the array pointed to by *rgcbReservation*. This parameter must be zero if *rgcbReservation* is **NULL** or the CLFS_FLAG_USE_RESERVATION flag of *fFlags* is set.

`rgcbReservation`

A pointer to an array of LONGLONG-typed variables. The caller sets each element of the array to the size, in bytes, of a record that must have space reserved for it. On return, each array element receives that actual size of the space reserved for the record. This includes the space required for headers and alignment. If the reservation value is

negative, a reserved record that most nearly matches the absolute value of the provided negative value will be freed. This parameter can be **NULL** if *cReserveRecords* is zero and must be **NULL** if the CLFS_FLAG_USE_RESERVATION flag of *fFlags* is set.

`fFlags`

This parameter can be zero or any combination of the following flags.

FLAG	MEANING
CLFS_FLAG_FORCE_APPEND	After the current record is appended to a log I/O block, the block is queued, in LSN sequence, to stable storage. This flag provides no guarantee that the record is forced to stable storage (see CLFS_FLAG_FORCE_FLUSH).
CLFS_FLAG_FORCE_FLUSH	After the current record is appended to a log I/O block, the block is forced to stable storage.
CLFS_FLAG_USE_RESERVATION	The current record is placed in reserved space in a log I/O block within the marshalling area. The number of reserved records in the marshalling area is reduced by one. If this flag is set, *cReserveRecords* must be zero and *rgcbReservation* must be **NULL**.

`plsn`

A pointer to a CLFS_LSN structure that receives the LSN of the appended record. This parameter can be **NULL** if *cWriteEntries* is zero.

Return Value

ClfsReserveAndAppendLog returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The **ClfsReserveAndAppendLog** routine changes its fundamental behavior based on the presence of optional parameters and the state of the CLFS_USE_RESERVATION flag. The following table summarizes common scenarios.

PARAMETER AND FLAG VALUES	ACTIONS PERFORMED
cWriteEntries = 0. *rgWriteEntries* = **NULL**. *plsn* = **NULL**.	Reserves space for a set of records, but does not append the records to the marshalling area. The *rgcbReservation* parameter gives the size of the data portion of each record that needs space reserved.
cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* = 0. *rgcbReservation* is **NULL**. CLFS_USE_RESERVATION is set.	Appends a record to the marshalling area by using space that has already been reserved. Reduces the number of reserved record spaces by one.
cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* = 0. *rgcbReservation* is **NULL**. CLFS_USE_RESERVATION is cleared.	Appends a record to the marshalling area by reserving new space. Leaves the number of reserved record spaces unchanged.

cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* > 0. rgcbReservation is not **NULL**. CLFS_USE_RESERVATION flag is cleared.	Appends a record to the marshalling area by reserving new space. Also reserves space for a set of records that are not appended at this time. The *rgcbReservation* parameter gives the size of each record that needs space reserved. Increases the number of reserved record spaces by the value of *cReserveRecords*.

Calling **ClfsReserveAndAppendLog** is equivalent to calling ClfsReserveAndAppendLogAligned with the *cbEntryAlignment* parameter set to one.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

CLFS_WRITE_ENTRY

ClfsCreateMarshallingArea

ClfsReserveAndAppendLogAligned

ClfsReserveAndAppendLogAligned function

1/25/2019 • 4 minutes to read • Edit Online

The **ClfsReserveAndAppendLogAligned** routine reserves space in a marshalling area or appends a record to a marshalling area or does both atomically. The record's data is aligned on specified boundaries.

Syntax

```
CLFSUSER_API NTSTATUS ClfsReserveAndAppendLogAligned(
  PVOID             pvMarshalContext,
  PCLFS_WRITE_ENTRY rgWriteEntries,
  ULONG             cWriteEntries,
  ULONG             cbEntryAlignment,
  PCLFS_LSN         plsnUndoNext,
  PCLFS_LSN         plsnPrevious,
  ULONG             cReserveRecords,
  PLONGLONG         rgcbReservation,
  ULONG             fFlags,
  PCLFS_LSN         plsn
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`rgWriteEntries`

A pointer to an array of CLFS_WRITE_ENTRY structures, each of which holds a pointer to a buffer of data that will become part of the record that is appended to the log. This parameter can be **NULL** if *cWriteEntries* is zero.

`cWriteEntries`

The number of elements in the array pointed to by *rgWriteEntries*. This parameter must be zero if *rgWriteEntries* is **NULL**.

`cbEntryAlignment`

The byte alignment of the data entries pointed to by *rgWriteEntries* as they are marshaled into a single record. A value of one specifies simple concatenation (see ClfsReserveAndAppendLog). A value larger than one can result in zeros being placed between entries in the record. The value of this parameter must be greater than zero.

`plsnUndoNext`

A pointer to a CLFS_LSN structure that supplies the undo-next LSN of the record to be appended.

`plsnPrevious`

A pointer to a CLFS_LSN structure that supplies the previous LSN of the record to be appended.

`cReserveRecords`

The number of elements in the array pointed to by *rgcbReservation*. This parameter must be zero if

rgcbReservation is **NULL** or the CLFS_FLAG_USE_RESERVATION flag of *fFlags* is set.

`rgcbReservation`

A pointer to an array of LONGLONG-typed variables. The caller sets each element of the array to the size, in bytes, of a record that must have space reserve for it. On return, each array element receives that actual size of the space reserved for the record. This includes the space required for headers and alignment. If the reservation value is negative, a reserved record that most nearly matches the absolute value of the provided negative value will be freed. This parameter can be **NULL** if *cReserveRecords* is zero and must be **NULL** if the CLFS_FLAG_USE_RESERVATION flag of *fFlags* is set.

`fFlags`

This parameter can be any combination of the following flags.

FLAG	MEANING
CLFS_FLAG_FORCE_APPEND	After the current record is appended to a log I/O block, the block is queued, in LSN sequence, to stable storage. This flag provides no guarantee that the record is forced to stable storage (see CLFS_FLAG_FORCE_FLUSH).
CLFS_FLAG_FORCE_FLUSH	After the current record is appended to a log I/O block, the block is forced to stable storage.
CLFS_FLAG_USE_RESERVATION	The current record is placed in reserved space in an I/O block. The number of reserved records in the marshalling area is reduced by one. If this flag is set, *cReserveRecords* must be zero and *rgcbReservation* must be **NULL**.

`plsn`

A pointer to a CLFS_LSN structure that receives the LSN of the appended record. This parameter can be **NULL** if *cWriteEntries* is zero.

Return Value

ClfsReserveAndAppendLogAligned returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The **ClfsReserveAndAppendLogAligned** routine changes its fundamental behavior based on the presence of optional parameters and the state of the CLFS_USE_RESERVATION flag. The following table summarizes common scenarios.

PARAMETER AND FLAG VALUES	ACTIONS PERFORMED
cWriteEntries = 0. *rgWriteEntries* = **NULL**. *plsn* = **NULL**.	Reserves space for a set of records, but does not append the records to the marshalling area. The *rgcbReservation* parameter gives the size of the data portion of each record that needs space reserved.
cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* = 0. *rgcbReservation* is **NULL**. CLFS_USE_RESERVATION is set.	Appends a record to the marshalling area by using space that has already been reserved. Reduces the number of reserved record spaces by one.

cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* = 0. *rgcbReservation* is **NULL**. CLFS_USE_RESERVATION is cleared.	Appends a record to the marshalling area by reserving new space. Leaves the number of reserved record spaces unchanged.
cWriteEntries > 0. *rgWriteEntries* is not **NULL**. *plsn* is not **NULL**. *cReserveRecords* > 0. *rgcbReservation* is not **NULL**. CLFS_USE_RESERVATION flag is cleared.	Appends a record to the marshalling area by reserving new space. Also reserves space for a set of records that are not appended at this time. The *rgcbReservation* parameter gives the size of each record that needs space reserved. Increases the number of reserved record spaces by the value of *cReserveRecords*.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

CLFS_WRITE_ENTRY

ClfsCreateMarshallingArea

ClfsReserveAndAppendLog

ClfsScanLogContainers function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsScanLogContainers** routine retrieves descriptive information for a sequence of containers that belong to a particular CLFS log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsScanLogContainers(
  PCLFS_SCAN_CONTEXT pcxScan,
  CLFS_SCAN_MODE     eScanMode
);
```

Parameters

`pcxScan`

A pointer to a CLFS_SCAN_CONTEXT structure. The caller previously allocated the structure and initialized it by calling ClfsCreateScanContext. In particular, the **pInfoContainer** member was initialized to point to an array of CLFS_CONTAINER_INFORMATION structures, and the **cContainers** member was initialized to the number of elements in the array. On return, the structures in the array receive descriptive information for the containers in the sequence.

`eScanMode`

A set of flags that specify the direction of the scan and whether the scan context should be closed. The following flags are available to callers of this routine.

VALUE	MEANING
CLFS_SCAN_FORWARD	Scan in the forward direction. For example, if container 5 is the most recently scanned container and *pxcScan*->**cContainers** is equal to two, this routine will return descriptors for containers 6 and 7.
CLFS_SCAN_BACKWARD	Scan in the backward direction. For example, if container 5 is the most recently scanned container and *pxcScan*->**cContainers** is equal to two, this routine will return descriptors for containers 4 and 3.
CLFS_SCAN_CLOSE	Free all resources associated with the scan context pointed to by *pcxScan*.

If the CLFS_SCAN_CLOSE flag is set, both the CLFS_SCAN_FORWARD and CLFS_SCAN_BACKWARD flags must be clear.

If the CLFS_SCAN_CLOSE flag is clear, only one of the direction flags (CLFS_SCAN_FORWARD or CLFS_SCAN_BACKWARD) must be set.

Return Value

ClfsScanLogContainers returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes

defined in Ntstatus.h.

Remarks

CLFS uses the CLFS_SCAN_CONTEXT structure to track where a container scan starts (the **cIndex** member) and how many containers are scanned in each call to **ClfsScanLogContainers**. The value N of the **cContainers** member specifies that each time **ClfsScanLogContainers** is called, the next N containers are scanned.

The **cContainersReturned** member of the **CLFS_SCAN_CONTEXT** structure receives the number of containers actually scanned in a single call to **ClfsScanLogContainers**.

When **ClfsScanLogContainers** returns STATUS_NO_MORE_ENTRIES, there are no more containers to be scanned.

When you have finished using the scan context pointed to by *pcxScan*, you must call **ClfsScanLogContainers**, with *eScanMode* equal to CLFS_SCAN_CLOSE, to free any resources associated with the scan context.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_CONTAINER_INFORMATION

CLFS_SCAN_CONTEXT

ClfsCreateScanContext

ClfsSetArchiveTail function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsSetArchiveTail** routine sets the archive tail of a CLFS log to a specified LSN.

Syntax

```
CLFSUSER_API NTSTATUS ClfsSetArchiveTail(
  PLOG_FILE_OBJECT plfoLog,
  PCLFS_LSN        plsnArchiveTail
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS log. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`plsnArchiveTail`

A pointer to a CLFS_LSN structure that specifies the LSN that is to become the new archive tail. This must be the exact LSN of a record in the log.

Return Value

ClfsSetArchiveTail returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The *plsnArchiveTail* value specifies where archiving starts in the log. The next archiving will start at or before this LSN.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys

IRQL	"<= APC_LEVEL"

See Also

ClfsAdvanceLogBase

ClfsCreateLogFile

ClfsSetEndOfLog

LOG_FILE_OBJECT

ClfsSetEndOfLog function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsSetEndOfLog** routine truncates a CLFS stream.

Syntax

```
CLFSUSER_API NTSTATUS ClfsSetEndOfLog(
  PLOG_FILE_OBJECT plfoLog,
  PCLFS_LSN        plsnEnd
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. This stream must be the only stream of a dedicated log. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`plsnEnd`

A pointer to a CLFS_LSN structure that supplies the LSN of the record that is to become the last record of the stream. This must be the exact LSN of one of the records in the stream.

Return Value

ClfsSetEndOfLog returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

The LSN supplied in the *plsnEnd* value is not validated. It is the responsibility of the caller to provide a valid LSN; that is, one that is the exact LSN of a record in the stream.

This stream represented by *plfoLog* must be from a dedicated log. This routine does not support streams from multiplexed logs.

For an explanation of CLFS concepts and terminology, see Common Log File System.

The kernel-mode routine **ClfsSetEndOfLog** is reserved for future use and currently always returns STATUS_NOT_SUPPORTED. However, the user-mode routine **SetEndOfLog** is currently supported.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop

Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

ClfsAdvanceLogBase

ClfsCreateLogFile

ClfsSetArchiveTail

ClfsSetEndOfLog

LOG_FILE_OBJECT

ClfsSetLogFileInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsSetLogFileInformation** routine sets metadata and state information for a specified stream and its underlying physical log.

Syntax

```
CLFSUSER_API NTSTATUS ClfsSetLogFileInformation(
  PLOG_FILE_OBJECT           plfoLog,
  CLFS_LOG_INFORMATION_CLASS eInformationClass,
  PVOID                      pinfoBuffer,
  ULONG                      cbBuffer
);
```

Parameters

`plfoLog`

A pointer to a LOG_FILE_OBJECT structure that represents a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateLogFile.

`eInformationClass`

A CLFS_LOG_INFORMATION_CLASS value that specifies the class of information being set.

`pinfoBuffer`

A pointer to a buffer that supplies the log information. The structure of this buffer varies according to the class of information specified by *eInformationClass*. The following table shows the relationship between the information class and the buffer type.

VALUE OF EINFORMATIONCLASS	TYPE OF BUFFER POINTED TO BY *PINFOBUFFER*
ClfsLogBasicInformation	CLFS_INFORMATION
ClfsLogBasicInformationPhysical	CLFS_INFORMATION
ClfsLogNameInformation	CLFS_LOG_NAME_INFORMATION
ClfsLogPhysicalNameInformation	CLFS_LOG_NAME_INFORMATION
ClfsLogStreamIdentifierInformation	CLFS_STREAM_ID_INFORMATION

`cbBuffer`

The size, in bytes, of the buffer pointed to by *pinfoBuffer*.

Return Value

ClfsSetLogFileInformation returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_INFORMATION

CLFS_LOG_INFORMATION_CLASS

CLFS_STREAM_ID_INFORMATION

ClfsCreateLogFile

ClfsQueryLogFileInformation

LOG_FILE_OBJECT

ClfsTerminateReadLog function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsTerminateReadLog** routine invalidates a specified read context after freeing resources associated with the context.

Syntax

```
CLFSUSER_API NTSTATUS ClfsTerminateReadLog(
  PVOID pvCursorContext
);
```

Parameters

`pvCursorContext`

A pointer to the read context to be invalidated. The caller previously obtained this pointer by calling ClfsReadLogRecord or ClfsReadRestartArea.

Return Value

ClfsTerminateReadLog returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

> **Warning** Failure to call this routine can lead to memory leaks, premature exhaustion of log I/O blocks, and increased frequency of log flushes.
>
> Attempting to use *pvCursorContext* after it is freed is equivalent to accessing freed memory and can lead to unexpected behavior.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys

IRQL	"<= APC_LEVEL"

See Also

ClfsReadLogRecord

ClfsReadRestartArea

ClfsWriteRestartArea function

1/25/2019 • 2 minutes to read • Edit Online

The **ClfsWriteRestartArea** routine atomically appends a new restart record to a CLFS stream, flushes the restart record to stable storage, and optionally updates the base LSN of the stream.

Syntax

```
CLFSUSER_API NTSTATUS ClfsWriteRestartArea(
  PVOID     pvMarshalContext,
  PVOID     pvRestartBuffer,
  ULONG     cbRestartBuffer,
  PCLFS_LSN plsnBase,
  ULONG     fFlags,
  PULONG    pcbWritten,
  PCLFS_LSN plsnNext
);
```

Parameters

`pvMarshalContext`

A pointer to an opaque context that represents a marshalling area associated with a CLFS stream. The caller previously obtained this pointer by calling ClfsCreateMarshallingArea.

`pvRestartBuffer`

A pointer to a buffer that contains the data for the restart record.

`cbRestartBuffer`

The size, in bytes, of the buffer pointed to by *pvRestartBuffer*. This is the size of the restart data.

`plsnBase`

A pointer to a CLFS_LSN structure that specifies a new base LSN for the stream. If this parameter is **NULL**, the base LSN is not changed.

`fFlags`

This parameter must be one of the following values.

VALUE	MEANING
0	The restart record is placed in newly allocated space in an I/O block. The number of reserved records in the marshalling area is not changed.
CLFS_FLAG_USE_RESERVATION	The restart record is placed in previously reserved space in an I/O block. The number of reserved records in the marshalling area is reduced by one.

`pcbWritten`

A pointer to a ULONG-typed variable that receives the number of bytes actually forced to stable storage. This

parameter can be **NULL**.

`plsnNext`

A pointer to a **CLFS_LSN** structure that receives the LSN of the newly written restart record.

Return Value

ClfsWriteRestartArea returns STATUS_SUCCESS if it succeeds; otherwise, it returns one of the error codes defined in Ntstatus.h.

Remarks

Typically ClfsWriteRestartArea is called as the last act of a client checkpoint.

ClfsWriteRestartArea is a relatively expensive operation because it causes a flush of all records currently in the marshalling area along with a flush of stream and log metadata.

At any one time, only one marshalling area should be used to write data to a stream. Having two marshalling areas writing into the stream might result in stream corruption.

If you just want to set the base LSN of a stream, use ClfsAdvanceLogBase, which does not necessarily flush any data to stable storage.

For an explanation of CLFS concepts and terminology, see Common Log File System.

Requirements

Minimum supported client	Available in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
Library	Clfs.lib
DLL	Clfs.sys
IRQL	"<= APC_LEVEL"

See Also

CLFS_LSN

ClfsAdvanceLogBase

ClfsCreateMarshallingArea

ClfsReadPreviousRestartArea

ClfsReadRestartArea

CM_POWER_DATA structure

1/25/2019 • 2 minutes to read • Edit Online

The **CM_POWER_DATA** structure contains information about a device's power management state and capabilities.

Syntax

```
typedef struct CM_Power_Data_s {
  ULONG              PD_Size;
  DEVICE_POWER_STATE PD_MostRecentPowerState;
  ULONG              PD_Capabilities;
  ULONG              PD_D1Latency;
  ULONG              PD_D2Latency;
  ULONG              PD_D3Latency;
  DEVICE_POWER_STATE PD_PowerStateMapping[POWER_SYSTEM_MAXIMUM];
  SYSTEM_POWER_STATE PD_DeepestSystemWake;
} CM_POWER_DATA, *PCM_POWER_DATA;
```

Members

`PD_Size`

The size, in bytes, of this structure.

`PD_MostRecentPowerState`

A DEVICE_POWER_STATE value representing the device's most recent power state.

`PD_Capabilities`

The device's power capabilities, represented by a combination of the following bit flags:

PDCAP_D0_SUPPORTED

PDCAP_D1_SUPPORTED

PDCAP_D2_SUPPORTED

PDCAP_D3_SUPPORTED

PDCAP_WAKE_FROM_D0_SUPPORTED

PDCAP_WAKE_FROM_D1_SUPPORTED

PDCAP_WAKE_FROM_D2_SUPPORTED

PDCAP_WAKE_FROM_D3_SUPPORTED

PDCAP_WARM_EJECT_SUPPORTED

`PD_D1Latency`

The device's latency when returning the D0 state from the D1 state. For more information, see the **D1Latency** member of DEVICE_CAPABILITIES.

`PD_D2Latency`

The device's latency when returning the D0 state from the D2 state. For more information, see the **D2Latency** member of **DEVICE_CAPABILITIES**.

`PD_D3Latency`

The device's latency when returning the D0 state from the D3 state. For more information, see the **D3Latency** member of **DEVICE_CAPABILITIES**.

`PD_PowerStateMapping`

An array of **DEVICE_POWER_STATE** values representing the maximum device power state achievable for each system power state. For more information, see the **DeviceState** member of DEVICE_CAPABILITIES.

`PD_DeepestSystemWake`

Specifies the least-powered system state from which the device can wake the system. **PD_DeepestSystemWake** typically indicates one of the system sleeping states, S1, S2, or S3 (as specified by **PowerSystemSleeping1**, **PowerSystemSleeping2**, and **PowerSystemSleeping3**, respectively). Note, however, that some devices might be able to wake the system from the system hibernate state, S4 (as specified by **PowerSystemHibernate**), or even from the system shutdown state, S5 (as specified by **PowerSystemShutdown**).

Requirements

Minimum supported client	Available in Microsoft Windows XP and later versions of the Windows operating system.
Header	wdm.h (include Ntddk.h, Ntpoapi.h)

See Also

DEVICE_CAPABILITIES

DEVICE_POWER_STATE

SetupDiGetDeviceRegistryProperty

CmCallbackGetKeyObjectID function

1/25/2019 • 2 minutes to read • Edit Online

The **CmCallbackGetKeyObjectID** routine retrieves the unique identifier and object name that are associated with a specified registry key object.

> **Note** Starting with Windows 8, registry filter drivers should call the CmCallbackGetKeyObjectIDEx routine instead of **CmCallbackGetKeyObjectID**. For more information, see Remarks.

Syntax

```
NTSTATUS CmCallbackGetKeyObjectID(
  PLARGE_INTEGER   Cookie,
  PVOID            Object,
  PULONG_PTR       ObjectID,
  PCUNICODE_STRING *ObjectName
);
```

Parameters

`Cookie`

The cookie value that the driver previously obtained by calling the CmRegisterCallback or CmRegisterCallbackEx routine.

`Object`

The pointer value that the driver's RegistryCallback callback routine received in the **Object** member of one of the **REG_*XXX*_KEY_INFORMATION** structures.

> **Warning** In certain circumstances registry callback notification structures may contain invalid non-NULL object pointers. Registry filtering drivers must not pass such pointers to this routine. For more information, see Invalid Key Object Pointers in Registry Notifications.

`ObjectID`

A pointer to a location that receives a pointer to the unique identifier that represents the registry key that *Object* specifies. This parameter is optional and can be **NULL**.

`ObjectName`

A pointer to a location that receives a pointer to a UNICODE_STRING structure. This structure contains the object name of the registry key object that *Object* specifies. The object name is actually the full path name of the registry key that the object represents. The caller must not write to this **UNICODE_STRING** structure or free it. This parameter is optional and can be **NULL**.

Return Value

CmCallbackGetKeyObjectID returns STATUS_SUCCESS if the operation succeeds. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ PAR AM ETE R	The *Cookie* or *Object* parameter is invalid.

Remarks

The **CmCallbackGetKeyObjectID** routine is available starting with Windows Vista. An improved version of this routine, CmCallbackGetKeyObjectIDEx, is available starting with Windows 8. Drivers that run only in Windows 8 and later versions of Windows should call **CmCallbackGetKeyObjectIDEx** instead of **CmCallbackGetKeyObjectID**.

Drivers can use **CmCallbackGetKeyObjectID** to obtain the registry key identifier, the object name, or both, by supplying non-**NULL** values for the *ObjectID* or *ObjectName* parameters.

After the driver has obtained the identifier or name, the identifier or name is valid until the driver's *RegistryCallback* routine receives pre-notification of handle closure.

The driver must not modify the object name.

If two registry key objects represent the same registry key, the key identifiers for both objects are identical.

For more information about **CmCallbackGetKeyObjectID** and registry filtering operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

CmCallbackGetKeyObjectIDEx

CmRegisterCallback

CmRegisterCallbackEx

RegistryCallback

UNICODE_STRING

CmCallbackGetKeyObjectIDEx function

1/25/2019 • 3 minutes to read • Edit Online

The **CmCallbackGetKeyObjectIDEx** routine retrieves the unique identifier and object name that are associated with a specified registry key object.

> **Note** This routine is available starting with Windows 8. In earlier versions of Windows, use the CmCallbackGetKeyObjectID routine instead. For more information, see Remarks.

Syntax

```
NTSTATUS CmCallbackGetKeyObjectIDEx(
  PLARGE_INTEGER   Cookie,
  PVOID            Object,
  PULONG_PTR       ObjectID,
  PCUNICODE_STRING *ObjectName,
  ULONG            Flags
);
```

Parameters

`Cookie`

A cookie value that represents the caller's registration to receive registry filter callbacks. The driver previously obtained this cookie value from the CmRegisterCallback or CmRegisterCallbackEx routine.

`Object`

A pointer to the registry key object. This parameter is the pointer value that the driver's RegistryCallback callback routine received in the **Object** member of one of the **REG_*XXX*_KEY_INFORMATION** structures.

> **Warning** In certain circumstances registry callback notification structures may contain invalid non-NULL object pointers. Registry filtering drivers must not pass such pointers to this routine. For more information, see Invalid Key Object Pointers in Registry Notifications.

`ObjectID`

A pointer to a location that receives a pointer to the key identifier for the registry key that *Object* represents. This identifier is unique across all keys in the registry. This parameter is optional and can be **NULL**. For more information, see Remarks.

`ObjectName`

A pointer to a location that receives a pointer to a UNICODE_STRING structure. This structure contains the object name of the registry key object that *Object* specifies. The object name is actually the full path name of the registry key that the object represents. The caller is responsible for freeing this structure by calling the CmCallbackReleaseKeyObjectIDEx routine. This parameter is optional and can be **NULL**. For more information, see Remarks.

`Flags`

Reserved. Set to zero.

Return Value

CmCallbackGetKeyObjectIDEx returns STATUS_SUCCESS if the operation succeeds. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *Cookie*, *Object*, or *Flags* parameter is invalid.

Remarks

Drivers can use **CmCallbackGetKeyObjectIDEx** to obtain the registry key identifier, the object name, or both, by supplying non-**NULL** values for the *ObjectID* or *ObjectName* parameters.

The caller can obtain the key identifier through the *ObjectID* parameter. If two registry key objects represent the same registry key, the key identifiers obtained from **CmCallbackGetKeyObjectIDEx** for the two objects are identical. If the name of the registry key changes, the key identifier obtained from **CmCallbackGetKeyObjectIDEx** does not change. The caller can use the key identifier to reliably track accesses that are made to a particular registry key through multiple key objects, and even across changes to the registry key name.

The caller can obtain the object name through the *ObjectName* parameter. The storage for the UNICODE_STRING structure that contains the object name remains valid until the caller calls CmCallbackReleaseKeyObjectIDEx to free the structure.

CmCallbackGetKeyObjectIDEx is an improved version of the CmCallbackGetKeyObjectID routine, which is available starting with Windows Vista. Drivers that run in versions of Windows earlier than Windows 8 should call **CmCallbackGetKeyObjectID** instead of **CmCallbackGetKeyObjectIDEx**. Drivers that run only in Windows 8 and later versions of Windows should call **CmCallbackGetKeyObjectIDEx** instead of **CmCallbackGetKeyObjectID**.

CmCallbackGetKeyObjectIDEx has two important features that are not available from **CmCallbackGetKeyObjectID**.

First, **CmCallbackGetKeyObjectIDEx** enables a registry filter driver to explicitly control the lifetime of the *ObjectName* structure. The driver can call **CmCallbackReleaseKeyObjectIDEx** at any time to free this structure. In contrast, the driver cannot explicitly free the *ObjectName* structure received from **CmCallbackGetKeyObjectID**. This structure is automatically released when all handles to the key are closed.

Second, if a registry filter driver calls **CmCallbackReleaseKeyObjectID** to get an *ObjectName* structure pointer, and, after this call, the name of the registry key changes, subsequent calls to **CmCallbackGetKeyObjectID** get a pointer to a stale copy of the *ObjectName* structure, which contains the old key name. The stale copy of this structure persists until all handles to the key are closed. In contrast, the *ObjectName* structure obtained from **CmCallbackGetKeyObjectIDEx** always contains the most up-to-date key name.

For more information about registry filter drivers, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

CmCallbackGetKeyObjectID

CmCallbackReleaseKeyObjectIDEx

CmRegisterCallback

CmRegisterCallbackEx

RegistryCallback

UNICODE_STRING

CmCallbackReleaseKeyObjectIDEx function

1/25/2019 • 2 minutes to read • Edit Online

The **CmCallbackReleaseKeyObjectIDEx** routine frees an object name string obtained from the CmCallbackGetKeyObjectIDEx routine.

Syntax

```
void CmCallbackReleaseKeyObjectIDEx(
  PCUNICODE_STRING ObjectName
);
```

Parameters

`ObjectName`

A pointer to a UNICODE_STRING structure that contains the object name string. The driver previously obtained this pointer by calling CmCallbackGetKeyObjectIDEx.

Return Value

None.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

CmCallbackGetKeyObjectIDEx

UNICODE_STRING

CmGetBoundTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **CmGetBoundTransaction** routine returns a pointer to the transaction object that represents the transaction, if any, that is associated with a specified registry key object.

Syntax

```
PVOID CmGetBoundTransaction(
  PLARGE_INTEGER Cookie,
  PVOID          Object
);
```

Parameters

`Cookie`

The cookie value that the driver previously obtain by calling CmRegisterCallback or CmRegisterCallbackEx.

`Object`

The pointer value that the driver's RegistryCallback callback routine received in one of the **REG_*XXX*_KEY_INFORMATION** structures.

Return Value

CmGetBoundTransaction returns a pointer to a transaction object, if the key object that the *Object* parameter specifies is associated with a transaction. Otherwise, the routine returns **NULL**.

Remarks

The **CmGetBoundTransaction** routine is available on Windows Vista and later versions of Windows.

For more information about **CmGetBoundTransaction** and filtering registry operations, see Filtering Registry Calls.

For more information about transactions, see Kernel Transaction Manager Design Guide.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

CmRegisterCallback

CmRegisterCallbackEx

RegistryCallback

CmGetCallbackVersion function

1/25/2019 • 2 minutes to read • Edit Online

The **CmGetCallbackVersion** routine retrieves the major and minor version numbers for the current version of the configuration manager's registry callback feature.

Syntax

```
void CmGetCallbackVersion(
  PULONG Major,
  PULONG Minor
);
```

Parameters

`Major`

A pointer to a location that receives the major version number.

`Minor`

A pointer to a location that receives the minor version number.

Return Value

None

Remarks

The **CmGetCallbackVersion** routine is available starting with Windows Vista.

For Windows Vista, the major version number is 1 and the minor version number is 0.

Starting with Windows 7, the major version number is 1 and the minor version number is 1.

Version 1.1 contains two changes from version 1.0.

First, in version 1.0, if multiple registry filter drivers are active on the computer at the same time, the **REG_POST_*XXX*_KEY_INFORMATION** structure passed to a driver's registry callback routine during the post-notification phase for a create-key or open-key operation might contain a non-NULL **Object** member, even though the operation failed and the **Status** member contains an error status. In version 1.1, the **Object** member is always NULL if the **Status** member is set to an error status value to indicate that the operation failed.

Second, in version 1.0, an uncaught exception in a registry callback routine is quietly accepted by the operating system. In version 1.1, this exception causes the computer to bug check.

For more information on the differences between versions, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

REG_POST_CREATE_KEY_INFORMATION

REG_POST_OPEN_KEY_INFORMATION

ZwCreateKey

ZwOpenKey

CmRegisterCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **CmRegisterCallback** routine is **obsolete** for Windows Vista and later operating system versions. Use CmRegisterCallbackEx instead.

The **CmRegisterCallback** routine registers a RegistryCallback routine.

Syntax

```
NTSTATUS CmRegisterCallback(
  PEX_CALLBACK_FUNCTION Function,
  PVOID                 Context,
  PLARGE_INTEGER        Cookie
);
```

Parameters

`Function`

A pointer to the RegistryCallback routine to register.

`Context`

A driver-defined value that the configuration manager will pass as the *CallbackContext* parameter to the *RegistryCallback* routine

`Cookie`

A pointer to a LARGE_INTEGER variable that receives the value that identifies the callback routine. When you unregister the callback routine, pass this value as the *Cookie* parameter to CmUnRegisterCallback.

Return Value

CmRegisterCallback returns STATUS_SUCCESS if the operation succeeds or the appropriate NTSTATUS error code if it fails.

Remarks

The **CmRegisterCallback** routine is available on Windows XP and later operating system versions. For Windows Vista and later operating system versions, you should use CmRegisterCallbackEx instead.

A driver calls **CmRegisterCallback** to register a RegistryCallback routine, which is called every time a thread performs an operation on the registry.

Call CmUnRegisterCallback to unregister a callback routine that **CmRegisterCallback** registered.

For more information about **CmRegisterCallback** and filtering registry operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

CmRegisterCallbackEx

CmUnRegisterCallback

RegistryCallback

CmRegisterCallbackEx function

1/25/2019 • 2 minutes to read • Edit Online

The **CmRegisterCallbackEx** routine registers a RegistryCallback routine.

Syntax

```
NTSTATUS CmRegisterCallbackEx(
  PEX_CALLBACK_FUNCTION Function,
  PCUNICODE_STRING      Altitude,
  PVOID                 Driver,
  PVOID                 Context,
  PLARGE_INTEGER        Cookie,
  PVOID                 Reserved
);
```

Parameters

`Function`

A pointer to the RegistryCallback routine to register.

`Altitude`

A pointer to a UNICODE_STRING structure. This structure must contain a string that represents the altitude of the calling minifilter driver. For more information, see Remarks.

`Driver`

A pointer to the DRIVER_OBJECT structure that represents the driver.

`Context`

A driver-defined value that the configuration manager will pass as the *CallbackContext* parameter to the *RegistryCallback* routine.

`Cookie`

A pointer to a LARGE_INTEGER variable that receives the value that identifies the callback routine. When you unregister the callback routine, pass this value as the *Cookie* parameter to CmUnRegisterCallback.

`Reserved`

This parameter is reserved for future use.

Return Value

CmRegisterCallbackEx returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION

STA TUS _FLT _IN STA NCE _AL TIT UDE _CO LLIS ION	The calling driver or another driver has already registered a RegistryCallback routine for the specified altitude.
STA TUS _IN SUF FICI ENT _RE SO URC ES	An attempt to allocate memory failed.

Remarks

The **CmRegisterCallbackEx** routine is available starting with Windows Vista.

A driver can call CmRegisterCallback or **CmRegisterCallbackEx** to register a RegistryCallback routine, which is called every time a thread performs an operation on the registry.

The *Altitude* parameter defines the position of the minifilter driver relative to other minifilters in the I/O stack when the minifilter is loaded. Allocation of altitudes to minifilters is managed by Microsoft. For more information about altitudes, see Load Order Groups and Altitudes for Minifilter Drivers.

Call CmUnRegisterCallback to unregister a callback routine that **CmRegisterCallbackEx** registered.

For more information about **CmRegisterCallbackEx** and filtering registry operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

CmRegisterCallback

CmUnRegisterCallback

DRIVER_OBJECT

RegistryCallback

UNICODE_STRING

CmSetCallbackObjectContext function

1/25/2019 • 2 minutes to read • Edit Online

The **CmSetCallbackObjectContext** routine associates specified context information with a specified registry object.

Syntax

```
NTSTATUS CmSetCallbackObjectContext(
  PVOID          Object,
  PLARGE_INTEGER Cookie,
  PVOID          NewContext,
  PVOID          *OldContext
);
```

Parameters

`Object`

A pointer to the registry key object that the driver is providing context information for. The driver obtains this pointer from the **ResultObject** member of one of the following structures:

REG_CREATE_KEY_INFORMATION REG_CREATE_KEY_INFORMATION_V1 REG_OPEN_KEY_INFORMATION REG_OPEN_KEY_INFORMATION_V1

`Cookie`

A pointer to a LARGE_INTEGER value that identifies the callback routine to associate the context with. The CmRegisterCallbackEx routine provided this value when you registered the callback routine.

`NewContext`

A pointer to driver-defined context information.

`OldContext`

A pointer to a location that receives a pointer to context information that the driver previously associated with the specified object and cookie. This parameter is optional and can be **NULL**.

Return Value

CmSetCallbackObjectContext returns STATUS_SUCCESS or another appropriate NTSTATUS-typed value.

Remarks

The **CmSetCallbackObjectContext** routine is available starting with Windows Vista.

A driver's RegistryCallback routine can call **CmSetCallbackObjectContext** for any registry key object after the object has been created or opened (that is, during a post-notification for a create operation, an open operation, or any subsequent notification up to the pre-notification of handle closure).

If a driver calls **CmSetCallbackObjectContext**, the driver's *RegistryCallback* routine will receive a **RegNtCallbackObjectContextCleanup** notification after the key object's handle has been closed or after the

driver calls CmUnRegisterCallback to unregister the *RegistryCallback* routine. When the *RegistryCallback* routine receives this notification, the routine should release any resources that it allocated for the object's context.

For more information about **CmSetCallbackObjectContext** and filtering registry operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

CmRegisterCallbackEx

CmUnRegisterCallback

REG_CREATE_KEY_INFORMATION

REG_CREATE_KEY_INFORMATION_V1

REG_OPEN_KEY_INFORMATION

REG_OPEN_KEY_INFORMATION_V1

RegistryCallback

CmUnRegisterCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **CmUnRegisterCallback** routine unregisters a RegistryCallback routine that a CmRegisterCallback or CmRegisterCallbackEx routine previously registered.

Syntax

```
NTSTATUS CmUnRegisterCallback(
  LARGE_INTEGER Cookie
);
```

Parameters

`Cookie`

A LARGE_INTEGER value that identifies the callback routine to unregister. **CmRegisterCallback** provided this value when you registered the callback routine.

Return Value

CmUnRegisterCallback returns STATUS_SUCCESS if it succeeds or the appropriate NTSTATUS error code if it fails. If the *Cookie* parameter does not match any registered callback routines, **CmUnRegisterCallback** returns STATUS_INVALID_PARAMETER.

Remarks

A driver that calls **CmRegisterCallback** or **CmRegisterCallbackEx** should call **CmUnRegisterCallback** before the driver is unloaded.

> **Note** A driver must not call **CmRegisterCallback** from its implementation of the RegistryCallback routine. That call will result in a deadlock.

For more information about **CmUnRegisterCallback** and filtering registry operations, see Filtering Registry Calls.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

CmRegisterCallback

CmRegisterCallbackEx

RegistryCallback

DMA_COMPLETION_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DmaCompletionRoutine* callback routine notifies the driver that previously requested a system DMA transfer that this transfer is complete.

Syntax

```
DMA_COMPLETION_ROUTINE DmaCompletionRoutine;

void DmaCompletionRoutine(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PVOID CompletionContext,
  DMA_COMPLETION_STATUS Status
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's system DMA channel.

`DeviceObject`

A pointer to a DEVICE_OBJECT structure. This structure is the physical device object (PDO) that represents the target device for the requested DMA operation.

`CompletionContext`

A pointer to a driver-determined context for the *DmaCompletionRoutine* routine. This context is the *CompletionContext* parameter value that the driver previously passed to the MapTransferEx call that initiated the system DMA transfer.

`Status`

The completion status of the DMA transfer. This parameter is a DMA_COMPLETION_STATUS enumeration value that indicates whether the DMA transfer completed successfully.

Return Value

None.

Remarks

As an option, a driver can implement a *DmaCompletionRoutine* routine. When the driver requests a system DMA transfer, the driver can supply a pointer to this routine with the request. After the DMA transfer completes, the *DmaCompletionRoutine* routine is automatically called to notify the driver.

A driver's *DmaCompletionRoutine* routine is called only if the DMA transfer requested by the driver uses a system DMA controller that can generate an interrupt after a DMA transfer completes. A driver that uses a bus-master

DMA device, or that uses a system DMA controller that does not generate interrupts, must use other means to determine when a DMA transfer completes. For example, this driver might use a timer interrupt to periodically poll the status of the DMA transfer.

A driver can supply a pointer to a *DmaCompletionRoutine* routine as an optional parameter to the MapTransferEx routine.

Examples

To define a *DmaCompletionRoutine* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DmaCompletionRoutine* callback routine that is named `MyDmaCompletionRoutine`, use the DMA_COMPLETION_ROUTINE type as shown in this code example:

```
DMA_COMPLETION_ROUTINE MyDmaCompletionRoutine;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDmaCompletionRoutine(
    PDMA_ADAPTER DmaAdapter,
    PDEVICE_OBJECT DeviceObject,
    PVOID CompletionContext,
    DMA_COMPLETION_STATUS Status
    )
  {
      // Function body
  }
```

The DMA_COMPLETION_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DMA_COMPLETION_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

See Also

BuildScatterGatherListEx

DEVICE_OBJECT

DMA_ADAPTER

DMA_COMPLETION_STATUS

DMA_OPERATIONS

GetScatterGatherListEx

MapTransferEx

DRIVER_ADD_DEVICE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *AddDevice* routine is responsible for creating functional device objects (FDO) or filter device objects (filter DO) for devices enumerated by the Plug and Play (PnP) manager.

Syntax

```
DRIVER_ADD_DEVICE DriverAddDevice;

NTSTATUS DriverAddDevice(
  _DRIVER_OBJECT *DriverObject,
  _DEVICE_OBJECT *PhysicalDeviceObject
)
{...}
```

Parameters

`*DriverObject`

Caller-supplied pointer to a DRIVER_OBJECT structure. This is the driver's driver object.

`*PhysicalDeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure representing a physical device object (PDO) created by a lower-level driver.

Return Value

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in Ntstatus.h.

Remarks

All kernel-mode drivers that support PnP must provide an *AddDevice* routine.

A driver's *AddDevice* routine should be named *Xxx* `AddDevice`, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *AddDevice* routine's address in **DriverObject->DriverExtension->AddDevice**.

An *AddDevice* routine's primary responsibilities are calling IoCreateDevice to create a device object, then calling IoAttachDeviceToDeviceStack to attach the device object to the device stack. For detailed information about implementing a driver's *AddDevice* routine, see Writing an AddDevice Routine.

An *AddDevice* routine runs in a system thread context at IRQL = PASSIVE_LEVEL.

Examples

To define an *AddDevice* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *AddDevice* callback routine that is named `MyAddDevice`, use the DRIVER_ADD_DEVICE

type as shown in this code example:

```
DRIVER_ADD_DEVICE MyAddDevice;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyAddDevice(
    struct _DRIVER_OBJECT  *DriverObject,
    struct _DEVICE_OBJECT  *PhysicalDeviceObject
    )
  {
      // Function body
  }
```

The DRIVER_ADD_DEVICE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_ADD_DEVICE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL (see Remarks section).

DRIVER_CANCEL callback function

1/25/2019 • 2 minutes to read • Edit Online

The *Cancel* routine cancels an I/O operation.

Syntax

```
DRIVER_CANCEL DriverCancel;

void DriverCancel(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp
)
{...}
```

Parameters

`*DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`*Irp`

Caller-supplied pointer to an IRP structure that describes the I/O operation to be canceled.

Return Value

None

Remarks

When a driver or other system component calls IoCancelIrp, the I/O manager calls the IRP's *Cancel* routine, if one has been registered for the IRP.

To register a *Cancel* routine for an IRP, a driver can use either of the following two methods:

1. A driver that provides a StartIo routine and uses the I/O manager-supplied device queue can specify a *Cancel* routine when calling IoStartPacket.
2. A driver that creates and manages supplemental device queues can register a *Cancel* routine by calling IoSetCancelRoutine.

Only one *Cancel* routine can be registered for an IRP at one time.

The I/O manager calls IoAcquireCancelSpinLock before calling a driver's *Cancel* routine, so the *Cancel* routine must call IoReleaseCancelSpinLock at some point. The routine should not hold the spin lock longer than necessary.

The *Cancel* routine executes in an arbitrary thread context at IRQL = DISPATCH_LEVEL until it calls **IoReleaseCancelSpinLock**, which changes the IRQL to a caller-supplied value. The driver should specify **Irp->CancelIrql** for this value.

The *Cancel* routine must set the I/O status block's **Status** member to STATUS_CANCELLED, and set its

Information member to zero. The routine must then complete the specified IRP by calling IoCompleteRequest.

For detailed information about implementing a driver's *Cancel* routine, see Canceling IRPs.

Examples

To define a *Cancel* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *Cancel* callback routine that is named `MyCancel`, use the DRIVER_CANCEL type as shown in this code example:

```
DRIVER_CANCEL MyCancel;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyCancel(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp
    )
  {
      // Function body
  }
```

The DRIVER_CANCEL function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_CANCEL function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL (see Remarks section).

DRIVER_CONTROL callback function

1/25/2019 • 2 minutes to read • Edit Online

This routine starts a DMA data-transfer or a data transfer operation.

Syntax

```
DRIVER_CONTROL DriverControl;

IO_ALLOCATION_ACTION DriverControl(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp,
  PVOID MapRegisterBase,
  PVOID Context
)
{...}
```

Parameters

`DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's *AddDevice* routine.

`Irp`

Caller-supplied pointer to an IRP structure. *Irp* is equal to the value of the **CurrentIrp** member of *DeviceObject* when the callback routine was registered.

`MapRegisterBase`

In the case of *AdapterControl*, this is a caller-supplied opaque value that represents the map registers that the system has assigned for this transfer operation. The driver passes this value to FlushAdapterBuffers, FreeMapRegisters, and MapTransfer.

In the case of *ControllerControl*, this is not used.

`Context`

Caller-supplied pointer to driver-defined context information, specified in a previous call to AllocateAdapterChannel.

Return Value

The routine must return one of the values defined by the IO_ALLOCATION_ACTION enumeration. Drivers of bus-master devices return either **DeallocateObject** or **DeallocateObjectKeepRegisters**; drivers that use system DMA return **KeepObject**.

Remarks

About implementing *AdapterControl*:

To register an *AdapterControl* routine for a specific device object, a driver must call IoGetDmaAdapter to obtain an

adapter object, then call AllocateAdapterChannel to request use of the adapter and to supply the *AdapterControl* routine's address. When the adapter is free, the system calls the *AdapterControl* routine.

If *AdapterControl* has been registered by a StartIo routine, then the *Irp* parameter is guaranteed to point to the IRP currently being processed by the *StartIo* routine. Otherwise, drivers must set the **CurrentIrp** member of the device object structure before calling **AllocateAdapterChannel**.

For detailed information about implementing an *AdapterControl* routine, see Writing AdapterControl Routines.

About implementing *ControllerControl*: A driver's *ControllerControl* routine executes in an arbitrary thread context at IRQL = DISPATCH_LEVEL.

To register a *ControllerControl* routine for a specific device object, a driver must call IoCreateController to obtain a controller object, then call IoAllocateController to request use of the controller and to supply the *ControllerControl* routine's address. When the controller is free, the system calls the *ControllerControl* routine.

For detailed information about implementing a *ControllerControl* routine, see Writing ControllerControl Routines. Also see Controller Objects.

The DRIVER_CONTROL function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_CONTROL function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Examples

To define a callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *AdapterControl* callback routine that is named `MyAdapterControl`, use the DRIVER_CONTROL type as shown in this code example:

```
DRIVER_CONTROL MyAdapterControl;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
IO_ALLOCATION_ACTION
 MyAdapterControl(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp,
    PVOID  MapRegisterBase,
    PVOID  Context
    )
  {
      // Function body
  }
```

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

See Also

AllocateAdapterChannel

DRIVER_DISPATCH callback function

1/25/2019 • 4 minutes to read • Edit Online

The callback routine services various IRPs. For a list of function codes, see Remarks.

Syntax

```
DRIVER_DISPATCH DriverDispatch;

NTSTATUS DriverDispatch(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp
)
{...}
```

Parameters

`*DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`*Irp`

Caller-supplied pointer to an IRP structure that describes the requested I/O operation.

Return Value

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in Ntstatus.h.

Remarks

Input parameters for all *Dispatch* routines are supplied in the IRP structure pointed to by *Irp*. Additional parameters are supplied in the driver's associated I/O stack location, which is described by the IO_STACK_LOCATION structure and can be obtained by calling IoGetCurrentIrpStackLocation.

Generally, all *Dispatch* routines execute in an arbitrary thread context at IRQL = PASSIVE_LEVEL, but there are exceptions. For more information, see Dispatch Routines and IRQLs.

For more information about dispatch routines, see Writing Dispatch Routines. For more information about IRPs, see Handling IRPs.

IRP	ABOUT IMPLEMENTING THE CALLBACK
IRP_MJ_CLEANUP	A driver's *DispatchCleanup* routine should be named *Xxx* DispatchCleanup, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchCleanup* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_CLEANUP].

IRP_MJ_CLOSE	A driver's *DispatchClose* routine should be named ***Xxx*DispatchClose**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchClose* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_CLOSE].
IRP_MJ_CREATE	A driver's *DispatchCreate* routine should be named *Xxx* DispatchCreate, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchCreate* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_CREATE].
IRP_MJ_CREATE or IRP_MJ_CLOSE	A driver can provide a single *DispatchCreateClose* routine instead of separate DispatchCreate and DispatchClose routines. A driver's *DispatchCreateClose* routine should be named ***Xxx*DispatchCreateClose**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchCreateClose* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_CREATE] and in *DriverObject*->**MajorFunction**[IRP_MJ_CLOSE].
IRP_MJ_DEVICE_CONTROL	A driver's *DispatchDeviceControl* routine should be named ***Xxx*DispatchDeviceControl**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchDeviceControl* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_DEVICE_CONTROL]. The system uses the FILE_*XXX* flags in the I/O control code to determine whether the IRP sender has the privileges to send the IRP to the device object. Drivers for Windows Server 2003 and later versions of Windows can use the IoValidateDeviceIoControlAccess routine to perform stricter access checks within *DispatchDeviceControl*.
IRP_MJ_FLUSH_BUFFERS	A driver's *DispatchFlushBuffers* routine should be named *Xxx* DispatchFlushBuffers, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchFlushBuffers* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_FLUSH_BUFFERS].
IRP_MJ_INTERNAL_DEVICE_CONTROL	A driver's *DispatchInternalDeviceControl* routine should be named ***Xxx*DispatchInternalDeviceControl**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchInternalDeviceControl* routine's address in *DriverObject*->**MajorFunction**[**IRP_MJ_INTERNAL_DEVICE_CONTROL**].
IRP_MJ_PNP	A driver's *DispatchPnP* routine should be named *Xxx* DispatchPnP, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchPnP* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_PNP].
IRP_MJ_POWER	A driver's *DispatchPower* routine should be named ***Xxx*DispatchPower**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchPower* routine's address in *DriverObject*->**MajorFunction**[**IRP_MJ_POWER**].

IRP_MJ_QUERY_INFORMATION	A driver's *DispatchQueryInformation* routine should be named ***Xxx*DispatchQueryInformation**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchQueryInformation* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_QUERY_INFORMATION].
IRP_MJ_READ	A driver's *DispatchRead* routine should be named ***Xxx*DispatchRead**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchRead* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_READ].
IRP_MJ_READ or IRP_MJ_WRITE	A driver can provide a single *DispatchReadWrite* routine instead of separate DispatchRead and DispatchWrite routines. A driver's *DispatchReadWrite* routine should be named ***Xxx*DispatchReadWrite**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchReadWrite* routine's address in *DriverObject*->**MajorFunction**[**IRP_MJ_READ**] and in *DriverObject*->**MajorFunction**[**IRP_MJ_WRITE**].
IRP_MJ_SET_INFORMATION	A driver's *DispatchSetInformation* routine should be named ***Xxx*DispatchSetInformation**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchSetInformation* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_SET_INFORMATION].
IRP_MJ_SHUTDOWN	A driver's *DispatchShutdown* routine should be named ***Xxx*DispatchShutdown**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchShutdown* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_SHUTDOWN]. Additionally, in order to receive **IRP_MJ_SHUTDOWN** requests, a driver must call IoRegisterShutdownNotification or IoRegisterLastChanceShutdownNotification to register its *DispatchShutdown* routine with the system.
IRP_MJ_SYSTEM_CONTROL	A driver's *DispatchSystemControl* routine should be named ***Xxx*DispatchSystemControl**, where ***Xxx*** is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchSystemControl* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_SYSTEM_CONTROL].
IRP_MJ_WRITE	A driver's *DispatchWrite* routine should be named ***Xxx*DispatchWrite**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *DispatchWrite* routine's address in *DriverObject*->**MajorFunction**[IRP_MJ_WRITE].

Examples

To define a callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DispatchCleanup* callback routine that is named `MyDispatchCleanup`, use the DRIVER_DISPATCH type as shown in this code example:

```
DRIVER_DISPATCH MyDispatchCleanup;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyDispatchCleanup(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp
    )
  {
      // Function body
  }
```

The DRIVER_DISPATCH function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_DISPATCH function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL (see Remarks section).

DRIVER_INITIALIZE callback function

1/25/2019 • 2 minutes to read • Edit Online

DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.

Syntax

```
DRIVER_INITIALIZE DriverInitialize;

NTSTATUS DriverInitialize(
  _DRIVER_OBJECT *DriverObject,
  PUNICODE_STRING RegistryPath
)
{...}
```

Parameters

`*DriverObject`

A pointer to a DRIVER_OBJECT structure. This is the driver's driver object.

`RegistryPath`

A pointer to a counted Unicode string specifying the path to the driver's registry key.

Return Value

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in Ntstatus.h.

Remarks

The *DriverObject* parameter supplies the **DriverEntry** routine with a pointer to the driver's driver object, which is allocated by the I/O manager. The **DriverEntry** routine must fill in the driver object with entry points for the driver's standard routines.

The *DriverObject* pointer gives the driver access to *DriverObject*->**HardwareDatabase**, which points to a counted Unicode string that specifies a path to the registry's **\Registry\Machine\Hardware** tree.

The registry path string pointed to by *RegistryPath* is of the form **\Registry\Machine\System\CurrentControlSet\Services<i>DriverName**. A driver can use this path to store driver-specific information; see Registry Keys for Drivers. The **DriverEntry** routine should save a copy of the Unicode string, not the pointer, since the I/O manager frees the *RegistryPath* buffer after **DriverEntry** returns.

For more information about implementing a **DriverEntry** routine, see Writing a DriverEntry Routine.

While it is possible to name this routine something other than **DriverEntry**, doing so is not recommended. The DDK-supplied build tools automatically inform the linker that the driver's entry point is called **DriverEntry**, so giving the routine another name requires you to modify the build tools. For more information about build tools, see Building a Driver.

Examples

To define a **DriverEntry** callback routine, you must first provide a function declaration that identifies the type of

callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

To define a **DriverEntry** callback routine, use the DRIVER_INITIALIZE type as shown in this code example:

```
DRIVER_INITIALIZE DriverEntry;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  DriverEntry(
    struct _DRIVER_OBJECT  *DriverObject,
    PUNICODE_STRING  RegistryPath
    )
  {
      // Function body
  }
```

The DRIVER_INITIALIZE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_INITIALIZE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Mcd.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

DRIVER_OBJECT

DRIVER_LIST_CONTROL callback function

1/25/2019 • 2 minutes to read • Edit Online

The *AdapterListControl* routine starts a direct memory access (DMA) scatter/gather operation.

Syntax

```
DRIVER_LIST_CONTROL DriverListControl;

void DriverListControl(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp,
  PSCATTER_GATHER_LIST ScatterGather,
  PVOID Context
)
{...}
```

Parameters

`DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`Irp`

Caller-supplied pointer to an IRP structure that describes the I/O operation, if the driver has a StartIo routine. Otherwise, not used.

`ScatterGather`

Caller-supplied pointer to a SCATTER_GATHER_LIST structure describing scatter/gather regions.

`Context`

Caller-supplied pointer to driver-defined context information, specified in a previous call to AllocateAdapterChannel.

Return Value

None

Remarks

To register an *AdapterListControl* routine for a specific device object, a driver must call IoGetDmaAdapter to obtain an adapter object, then call GetScatterGatherList to request use of the adapter and to supply the *AdapterListControl* routine's address. When the adapter is free, the system calls the *AdapterListControl* routine.

Examples

To define an *AdapterListControl* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *AdapterListControl* callback routine that is named `MyAdapterListControl`, use the DRIVER_LIST_CONTROL type as shown in this code example:

```
DRIVER_LIST_CONTROL MyAdapterListControl;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyAdapterListControl(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp,
    PSCATTER_GATHER_LIST  ScatterGather,
    PVOID  Context
    )
  {
      // Function body
  }
```

The DRIVER_LIST_CONTROL function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_LIST_CONTROL function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior. For detailed information about implementing an *AdapterListControl* routine, see Using Scatter/Gather DMA.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

DRIVER_STARTIO callback function

1/25/2019 • 2 minutes to read • Edit Online

The *StartIo* routine starts the I/O operation described by an IRP.

Syntax

```
DRIVER_STARTIO DriverStartio;

void DriverStartio(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp
)
{...}
```

Parameters

`*DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`*Irp`

Caller-supplied pointer to an IRP structure that describes the requested I/O operation.

Return Value

None

Remarks

A driver's *StartIo* routine executes in an arbitrary thread context at IRQL = DISPATCH_LEVEL.

The *StartIo* routine is optional. A driver's *StartIo* routine, if supplied, should be named *Xxx***StartIo**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *StartIo* routine's address in **DriverObject->DriverStartIo**. (If no routine is supplied, this pointer must be **NULL**.)

A driver can use IoSetStartIoAttributes to set attributes on when its *StartIo* routine is called.

For detailed information about implementing a driver's *StartIo* routine, see Writing a StartIo Routine.

Examples

To define a *StartIo* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *StartIo* callback routine that is named `MyStartIo`, use the DRIVER_STARTIO type as shown in this code example:

```
```

```
DRIVER_STARTIO MyStartIo;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyStartIo(
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp
    )
  {
      // Function body
  }
```

The DRIVER_STARTIO function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_STARTIO function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

DRIVER_UNLOAD callback function

1/25/2019 • 2 minutes to read • Edit Online

The *Unload* routine performs any operations that are necessary before the system unloads the driver.

Syntax

```
DRIVER_UNLOAD DriverUnload;

void DriverUnload(
  _DRIVER_OBJECT *DriverObject
)
{...}
```

Parameters

`*DriverObject`

Caller-supplied pointer to a DRIVER_OBJECT structure. This is the driver's driver object.

Return Value

None

Remarks

A driver's *Unload* routine executes in a system thread context at IRQL = PASSIVE_LEVEL.

The *Unload* routine is required for WDM drivers and optional for non-WDM drivers. A driver's *Unload* routine, if supplied, should be named *Xxx***Unload**, where *Xxx* is a driver-specific prefix. The driver's DriverEntry routine must store the *Unload* routine's address in **DriverObject->DriverUnload**. (If no routine is supplied, this pointer must be **NULL**.)

For detailed information about implementing a driver's *Unload* routine, see Writing an Unload Routine.

Examples

To define an *Unload* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *Unload* callback routine that is named `MyUnload`, use the DRIVER_UNLOAD type as shown in this code example:

```
DRIVER_UNLOAD MyUnload;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyUnload(
    struct _DRIVER_OBJECT  *DriverObject
    )
  {
      // Function body
  }
```

The DRIVER_UNLOAD function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_UNLOAD function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

EX_CALLBACK_FUNCTION callback function

1/25/2019 • 5 minutes to read • Edit Online

A filter driver's *RegistryCallback* routine can monitor, block, or modify a registry operation.

Syntax

```
EX_CALLBACK_FUNCTION ExCallbackFunction;

NTSTATUS ExCallbackFunction(
  PVOID CallbackContext,
  PVOID Argument1,
  PVOID Argument2
)
{...}
```

Parameters

`CallbackContext`

The value that the driver passed as the *Context* parameter to CmRegisterCallback or CmRegisterCallbackEx when it registered this *RegistryCallback* routine.

`Argument1`

A REG_NOTIFY_CLASS-typed value that identifies the type of registry operation that is being performed and whether the *RegistryCallback* routine is being called before or after the registry operation is performed.

`Argument2`

A pointer to a structure that contains information that is specific to the type of registry operation. The structure type depends on the REG_NOTIFY_CLASS-typed value for *Argument1*, as shown in the following table. For information about which REG_NOTIFY_CLASS-typed values are available for which operating system versions, see REG_NOTIFY_CLASS.

REG_NOTIFY_CLASS VALUE	STRUCTURE TYPE
RegNtDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPreSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPostSetValueKey	REG_POST_OPERATION_INFORMATION
RegNtDeleteValueKey	REG_DELETE_VALUE_KEY_INFORMATION

REG_NOTIFY_CLASS VALUE	STRUCTURE TYPE
RegNtPreDeleteValueKey	REG_DELETE_VALUE_KEY_INFORMATION
RegNtPostDeleteValueKey	REG_POST_OPERATION_INFORMATION
RegNtSetInformationKey	REG_SET_INFORMATION_KEY_INFORMATION
RegNtPreSetInformationKey	REG_SET_INFORMATION_KEY_INFORMATION
RegNtPostSetInformationKey	REG_POST_OPERATION_INFORMATION
RegNtRenameKey	REG_RENAME_KEY_INFORMATION
RegNtPreRenameKey	REG_RENAME_KEY_INFORMATION
RegNtPostRenameKey	REG_POST_OPERATION_INFORMATION
RegNtEnumerateKey	REG_ENUMERATE_KEY_INFORMATION
RegNtPreEnumerateKey	REG_ENUMERATE_KEY_INFORMATION
RegNtPostEnumerateKey	REG_POST_OPERATION_INFORMATION
RegNtEnumerateValueKey	REG_ENUMERATE_VALUE_KEY_INFORMATION
RegNtPreEnumerateValueKey	REG_ENUMERATE_VALUE_KEY_INFORMATION
RegNtPostEnumerateValueKey	REG_POST_OPERATION_INFORMATION
RegNtQueryKey	REG_QUERY_KEY_INFORMATION
RegNtPreQueryKey	REG_QUERY_KEY_INFORMATION
RegNtPostQueryKey	REG_POST_OPERATION_INFORMATION
RegNtQueryValueKey	REG_QUERY_VALUE_KEY_INFORMATION
RegNtPreQueryValueKey	REG_QUERY_VALUE_KEY_INFORMATION
RegNtPostQueryValueKey	REG_POST_OPERATION_INFORMATION
RegNtQueryMultipleValueKey	REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION
RegNtPreQueryMultipleValueKey	REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION
RegNtPostQueryMultipleValueKey	REG_POST_OPERATION_INFORMATION
RegNtPreCreateKey	REG_PRE_CREATE_KEY_INFORMATION
RegNtPreCreateKeyEx	REG_CREATE_KEY_INFORMATION**

REG_NOTIFY_CLASS VALUE	STRUCTURE TYPE
RegNtPostCreateKey	REG_POST_CREATE_KEY_INFORMATION
RegNtPostCreateKeyEx	REG_POST_OPERATION_INFORMATION
RegNtPreOpenKey	REG_PRE_OPEN_KEY_INFORMATION**
RegNtPreOpenKeyEx	REG_OPEN_KEY_INFORMATION
RegNtPostOpenKey	REG_POST_OPEN_KEY_INFORMATION
RegNtPostOpenKeyEx	REG_POST_OPERATION_INFORMATION
RegNtKeyHandleClose	REG_KEY_HANDLE_CLOSE_INFORMATION
RegNtPreKeyHandleClose	REG_KEY_HANDLE_CLOSE_INFORMATION
RegNtPostKeyHandleClose	REG_POST_OPERATION_INFORMATION
RegNtPreFlushKey	REG_FLUSH_KEY_INFORMATION
RegNtPostFlushKey	REG_POST_OPERATION_INFORMATION
RegNtPreLoadKey	REG_LOAD_KEY_INFORMATION
RegNtPostLoadKey	REG_POST_OPERATION_INFORMATION
RegNtPreUnLoadKey	REG_UNLOAD_KEY_INFORMATION
RegNtPostUnLoadKey	REG_POST_OPERATION_INFORMATION
RegNtPreQueryKeySecurity	REG_QUERY_KEY_SECURITY_INFORMATION
RegNtPostQueryKeySecurity	REG_POST_OPERATION_INFORMATION
RegNtPreSetKeySecurity	REG_SET_KEY_SECURITY_INFORMATION
RegNtPostSetKeySecurity	REG_POST_OPERATION_INFORMATION
RegNtCallbackObjectContextCleanup	REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION
RegNtPreRestoreKey	REG_RESTORE_KEY_INFORMATION
RegNtPostRestoreKey	REG_RESTORE_KEY_INFORMATION
RegNtPreSaveKey	REG_SAVE_KEY_INFORMATION
RegNtPostSaveKey	REG_SAVE_KEY_INFORMATION
RegNtPreReplaceKey	REG_REPLACE_KEY_INFORMATION

REG_NOTIFY_CLASS VALUE	STRUCTURE TYPE
RegNtPostReplaceKey	REG_REPLACE_KEY_INFORMATION
RegNtPreQueryKeyName	REG_QUERY_KEY_NAME
RegNtPostQueryKeyName	REG_POST_OPERATION_INFORMATION

** Starting with Windows 7, the actual data structure passed in when the notify class is RegNtPreCreateKeyEx or RegNtPreOpenKeyEx is the V1 version of this structure, REG_CREATE_KEY_INFORMATION_V1 or REG_OPEN_KEY_INFORMATION_V1 , respectively. Check the Reserved member to determine the version of the structure.

VERSION NUMBER	STRUCTURE NAME
0	REG_CREATE_KEY_INFORMATION and REG_OPEN_KEY_INFORMATION
1	REG_CREATE_KEY_INFORMATION_V1 and REG_OPEN_KEY_INFORMATION_V1

Return Value

For more information about when a *RegistryCallback* routine should return each of these status values, see Filtering Registry Calls.

Remarks

To be notified of registry operations, a kernel-mode component (such as the driver component of an antivirus software package) can call CmRegisterCallback or CmRegisterCallbackEx to register a *RegistryCallback* routine.

The *RegistryCallback* routine can inspect the contents of the input and output buffers that are supplied for registry operations. A registry operation can be initiated by a user-mode application that calls a user-mode registry routine (such as RegCreateKeyEx or RegOpenKeyEx) or by a driver that calls a kernel-mode registry routine (such as ZwCreateKey or ZwOpenKey). An *input buffer* is a memory buffer supplied by the initiator from which the registry reads input data for the operation. An *output buffer* is a buffer supplied by the initiator into which the registry writes output data requested by the initiator.

Before calling the *RegistryCallback* routine, the kernel probes (to verify alignment and accessibility) all members of the *Argument2* structures that point to output buffers in user-mode memory, but does not capture user-mode output buffers in system memory. The callback routine must enclose any access of an output buffer in a **try**/**except** block. If the callback routine needs to pass an output buffer pointer to a system routine (for example, ZwOpenKey), and the buffer is in user-mode memory, the callback routine must first capture the buffer.

The handling of input buffers depends on the Windows version. Starting with Windows 8, the kernel captures all input buffers pointed to by members of the *Argument2* structures in system memory before calling the *RegistryCallback* routine. In versions of Windows before Windows 8, the kernel probes all members of the *Argument2* structures that point to input buffers in user-mode memory, but captures only some of these buffers in system memory. In these earlier versions of Windows, the callback routine must enclose any access of an input buffer in a **try**/**except** block. Additionally, if the callback routine needs to pass an input buffer pointer to a system routine (for example, **ZwOpenKey**), and the buffer is in user-mode memory, the callback routine must first capture the buffer.

The following table summarizes the requirements for buffer accesses by the *RegistryCallback* routine.

BUFFER TYPE	WINDOWS VERSION	BUFFER POINTER PASSED TO CALLBACK ROUTINE	SAFE FOR CALLBACK ROUTINE TO DIRECTLY ACCESS?	SAFE TO PASS TO SYSTEM ROUTINES (SUCH AS ZWOPENKEY)?
User-mode input	Windows 8 and later	Points to captured data.	Yes	Yes
User-mode input	Windows 7 and earlier	Points to captured data or original user-mode buffer.	No. Must read under try/except.	No. Must allocate kernel memory, copy data from the original buffer under try/except, and pass the copied data to the system routine.
User-mode output	All	Points to original user-mode buffer.	No. Must write under try/except.	No. Must allocate kernel memory, pass kernel memory to the system routine, and copy the results back to the original buffer under try/except.
Kernel-mode input and output	All	Points to original kernel-mode buffer.	Yes	Yes

For more information about *RegistryCallback* routines and registry filter drivers, see Filtering Registry Calls.

A *RegistryCallback* executes at IRQL = PASSIVE_LEVEL and in the context of the thread that is performing the registry operation.

Examples

To define a *RegistryCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *RegistryCallback* callback routine that is named `MyRegistryCallback`, use the EX_CALLBACK_FUNCTION type as shown in this code example:

```
EX_CALLBACK_FUNCTION MyRegistryCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyRegistryCallback(
    PVOID  CallbackContext,
    PVOID  Argument1,
    PVOID  Argument2
    )
  {
      // Function body
  }
```

The EX_CALLBACK_FUNCTION function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the EX_CALLBACK_FUNCTION function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows XP (see Return Value section).
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL (see Remarks section).

See Also

CmRegisterCallback

CmUnRegisterCallback

ProbeForRead

REG_NOTIFY_CLASS

ZwOpenKey

ExAcquireResourceExclusiveLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireResourceExclusiveLite** routine acquires the given resource for exclusive access by the calling thread.

Syntax

```
BOOLEAN ExAcquireResourceExclusiveLite(
  PERESOURCE Resource,
  BOOLEAN    Wait
);
```

Parameters

`Resource`

A pointer to the resource to acquire.

`Wait`

Specifies the routine's behavior whenever the resource cannot be acquired immediately. If**TRUE**, the caller is put into a wait state until the resource can be acquired. If **FALSE**, the routine immediately returns, regardless of whether the resource can be acquired.

Return Value

ExAcquireResourceExclusiveLite returns **TRUE** if the resource is acquired. This routine returns **FALSE** if the input *Wait* is **FALSE** and exclusive access cannot be granted immediately.

Remarks

The following list describes whether and when a caller is given exclusive access to a given resource:

- If the resource is currently not owned, exclusive access is granted immediately to the current thread.
- If the caller already had acquired the resource for exclusive access, the current thread is granted the same type of access recursively.
- If the caller has shared access to the resource, the caller must release the lock before it attempts to reacquire it exclusively.
- If the resource is currently owned as exclusive by another thread, or if the caller only has shared access to the resource, the current thread is put into a wait state until the resource can be acquired.

> **Note** If two threads each hold a shared lock on the same resource and both attempt to acquire the lock exclusively without releasing their shared lock, they will deadlock. This means that each thread will wait for the other to release its shared hold on the lock, and neither will release its shared hold until the other does.

The caller can release the resource by calling either ExReleaseResourceLite or ExReleaseResourceForThreadLite.

Normal kernel APC delivery must be disabled before calling this routine. Disable normal kernel APC delivery by

calling KeEnterCriticalRegion. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling KeLeaveCriticalRegion. For more information, see Disabling APCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	ExclusiveResourceAccess, IrqlExApcLte3, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExAcquireResourceSharedLite

ExGetExclusiveWaiterCount

ExGetSharedWaiterCount

ExInitializeResourceLite

ExIsResourceAcquiredExclusiveLite

ExReinitializeResourceLite

ExReleaseResourceForThreadLite

KeEnterCriticalRegion

ExAcquireResourceSharedLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireResourceSharedLite** routine acquires the given resource for shared access by the calling thread.

Syntax

```
BOOLEAN ExAcquireResourceSharedLite(
  PERESOURCE Resource,
  BOOLEAN    Wait
);
```

Parameters

`Resource`

A pointer to the resource to acquire.

`Wait`

Specifies the routine's behavior whenever the resource cannot be acquired immediately. If**TRUE**, the caller is put into a wait state until the resource can be acquired. If **FALSE**, the routine immediately returns, regardless of whether the resource can be acquired.

Return Value

The caller can release the resource by calling either ExReleaseResourceLite or ExReleaseResourceForThreadLite.

ExAcquireResourceSharedLite returns **TRUE** if (or when) the resource is acquired. This routine returns **FALSE** if the input *Wait* is **FALSE** and shared access cannot be granted immediately.

Remarks

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.
- If the caller already has acquired the resource (for shared or exclusive access), the current thread is granted the same type of access recursively. Note that making this call does not convert a caller's exclusive access of a given resource to shared access.
- If the resource is currently owned as shared by another thread and no thread is waiting for exclusive access to the resource, shared access is granted to the caller immediately. The caller is put into a wait state if there is an exclusive waiter.
- If the resource is currently owned as exclusive by another thread or if there is another thread waiting for exclusive access and the caller does not already have shared access to the resource, the current thread either is put into a wait state (*Wait* set to **TRUE**) or **ExAcquireResourceSharedLite** returns **FALSE**.

Normal kernel APC delivery must be disabled before calling this routine. Disable normal kernel APC delivery by calling KeEnterCriticalRegion. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling KeLeaveCriticalRegion. For more information, see Disabling APCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte3, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExAcquireResourceExclusiveLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExConvertExclusiveToSharedLite

ExGetExclusiveWaiterCount

ExGetSharedWaiterCount

ExInitializeResourceLite

ExIsResourceAcquiredSharedLite

ExReinitializeResourceLite

ExReleaseResourceForThreadLite

ExAcquireRundownProtection function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireRundownProtection** routine tries to acquire run-down protection on a shared object so the caller can safely access the object.

Syntax

```
BOOLEAN ExAcquireRundownProtection(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

Return Value

ExAcquireRundownProtection returns **TRUE** if the routine successfully acquires run-down protection for the caller. Otherwise, it returns **FALSE**. A return value of **FALSE** indicates that the run down of the object has started and that the object must be treated as invalid.

Remarks

To safely access a shared object, a driver calls **ExAcquireRundownProtection** to acquire run-down protection on the object. The routine returns **TRUE** to indicate that run-down protection is in effect. When run-down protection is in effect, the driver can safely access the object without risk that the object will be deleted before the access completes.

After completing the access, the driver calls ExReleaseRundownProtection to release the previously acquired run-down protection.

If **ExAcquireRundownProtection** returns **FALSE**, the object is no longer accessible. For example, if the object is to be replaced with a new one, the caller must wait for the object owner to create the new object, and then negotiate access to the new object.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal

Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

EX_RUNDOWN_REF

ExInitializeRundownProtection

ExReleaseRundownProtection

ExAcquireRundownProtectionEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireRundownProtectionEx** routine tries to acquire run-down protection on a shared object so the caller can safely access the object.

Syntax

```
BOOLEAN ExAcquireRundownProtectionEx(
  PEX_RUNDOWN_REF RunRef,
  ULONG           Count
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

`Count`

The amount by which to increment to the run-down instance count of the object. The count is incremented only if the routine returns **TRUE**. For more information, see Remarks

Return Value

ExAcquireRundownProtectionEx returns **TRUE** if the routine successfully acquires run-down protection for the caller. Otherwise, it returns **FALSE**. A return value of **FALSE** indicates that the run down of the object has started and that the object must be treated as invalid.

Remarks

To safely access a shared object, a driver calls **ExAcquireRundownProtectionEx** to acquire run-down protection on the object.

The *RunRef* parameter points to an EX_RUNDOWN_REF structure that tracks the run-down status of the associated object. This status information includes a count of instances of run-down protection that are currently in effect on the object. The **ExAcquireRundownProtectionEx** and ExReleaseRundownProtectionEx routines increment and decrement this count by arbitrary amounts. Two related routines, ExAcquireRundownProtection and ExReleaseRundownProtection, increment and decrement this count by one.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows Server 2003.

Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExInitializeRundownProtection

ExReleaseRundownProtection

ExReleaseRundownProtectionEx

ExAcquireSharedStarveExclusive function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireSharedStarveExclusive** routine acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.

Syntax

```
BOOLEAN ExAcquireSharedStarveExclusive(
  PERESOURCE Resource,
  BOOLEAN    Wait
);
```

Parameters

`Resource`

A pointer to the resource to be acquired for shared access.

`Wait`

Specifies the routine's behavior whenever the resource cannot be acquired immediately. If**TRUE**, the caller is put into a wait state until the resource can be acquired. If **FALSE**, the routine immediately returns, regardless of whether the resource can be acquired.

Return Value

The caller can release the resource by calling either ExReleaseResourceLite or ExReleaseResourceForThreadLite.

ExAcquireSharedStarveExclusive returns **TRUE** if the requested access is granted. This routine returns **FALSE** if the input *Wait* is **FALSE** and shared access cannot be granted immediately.

Remarks

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.
- If the caller already has acquired the resource (for shared or exclusive access), the current thread is granted the same type of access recursively. Note that making this call does not convert a caller's exclusive access of a given resource to shared access.
- If the resource is currently owned as shared by another thread, shared access is granted to the caller immediately, even if another thread is waiting for exclusive access to that resource.
- If the resource is currently owned as exclusive by another thread, the caller either is put into a wait state (*Wait* set to **TRUE**) or **ExAcquireSharedStarveExclusive** returns **FALSE**.

Callers of **ExAcquireSharedStarveExclusive** usually need quick access to a shared resource in order to save an exclusive accessor from doing redundant work. For example, a file system might call this routine to modify a cached resource, such as a BCB pinned in the cache, before the cache manager can acquire exclusive access to the resource and write the cache out to disk.

Normal kernel APC delivery must be disabled before calling this routine. Disable normal kernel APC delivery by calling KeEnterCriticalRegion. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling KeLeaveCriticalRegion. For more information, see Disabling APCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte3, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExAcquireResourceSharedLite

ExAcquireSharedWaitForExclusive

ExConvertExclusiveToSharedLite

ExGetExclusiveWaiterCount

ExIsResourceAcquiredExclusiveLite

ExIsResourceAcquiredSharedLite

ExAcquireSharedWaitForExclusive function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAcquireSharedWaitForExclusive** routine acquires the given resource for shared access if shared access can be granted and there are no exclusive waiters.

Syntax

```
BOOLEAN ExAcquireSharedWaitForExclusive(
  PERESOURCE Resource,
  BOOLEAN    Wait
);
```

Parameters

`Resource`

A pointer to the resource to be acquired for shared access.

`Wait`

Specifies the routine's behavior whenever the resource cannot be acquired immediately. If**TRUE**, the caller is put into a wait state until the resource can be acquired. If **FALSE**, the routine immediately returns, regardless of whether the resource can be acquired.

Return Value

ExAcquireSharedWaitForExclusive returns **TRUE** if the requested access is granted or an exclusive owner releases the resource. This routine returns **FALSE** if the input *Wait* is **FALSE** and shared access cannot be granted immediately.

Remarks

Most drivers should use ExAcquireResourceSharedLite instead of **ExAcquireSharedWaitForExclusive**.

The caller can release the resource by calling either ExReleaseResourceLite or ExReleaseResourceForThreadLite.

If shared access cannot be granted immediately, the caller can wait for other threads to acquire and release exclusive ownership of the resource.

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.
- If the caller already has exclusive access to the resource, the current thread is granted the same type of access recursively.
- If the resource is currently owned as shared and there are no pending attempts to acquire exclusive access, shared access is granted to the caller immediately.
- If the resource is currently owned as shared but there is a pending attempt to acquire exclusive access, the caller either is put into a wait state (*Wait* set to **TRUE**) or **ExAcquireSharedWaitForExclusive** returns **FALSE**.

When the current thread waits to acquire the resource until after a pending exclusive ownership has been released, **ExAcquireSharedWaitForExclusive** returns **TRUE** when the current thread is granted shared access to the resource and resumes execution.

The behavior of **ExAcquireSharedWaitForExclusive** is identical to that of ExAcquireResourceSharedLite unless the calling thread already owns the resource as shared and there are exclusive waiters. In that case, **ExAcquireSharedWaitForExclusive** allows the exclusive waiters to acquire exclusive ownership of the resource.

If the caller specifies **TRUE** for the *Wait* parameter, the caller blocks until another thread frees the resource on behalf of the caller, using ExReleaseResourceForThread. Driver writers must be careful to ensure that another thread actually releases the resource; otherwise the caller is deadlocked. **ExAcquireResourceSharedLite** does not have this property, so drivers should use that routine unless they require the particular behavior of **ExAcquireSharedWaitForExclusive**.

Normal kernel APC delivery must be disabled before calling this routine. Disable normal kernel APC delivery by calling KeEnterCriticalRegion. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling KeLeaveCriticalRegion. For more information, see Disabling APCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte3, WithinCriticalRegion, HwStorPortProhibitedDDIs, SpNoWait, WithinCriticalRegion(storport)

See Also

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExConvertExclusiveToSharedLite

ExGetExclusiveWaiterCount

ExIsResourceAcquiredExclusiveLite

ExIsResourceAcquiredSharedLite

ExReleaseResourceForThread

ExAllocateFromLookasideListEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAllocateFromLookasideListEx** routine removes the first entry from the specified lookaside list, or, if the list is empty, dynamically allocates the storage for a new entry.

Syntax

```
PVOID ExAllocateFromLookasideListEx(
  PLOOKASIDE_LIST_EX Lookaside
);
```

Parameters

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes a lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

Return Value

ExAllocateFromLookasideListEx returns a pointer to a lookaside-list entry, if an entry is available in the list or can be dynamically allocated. Otherwise, this routine returns **NULL**.

Remarks

This routine removes the first entry, if an entry is available, from the specified lookaside list and returns a pointer to this entry. If the list is empty, the routine allocates storage for a new entry and returns a pointer to this entry. If this allocation fails, the routine returns **NULL**.

If the lookaside list is empty, **ExAllocateFromLookasideListEx** calls the LookasideListAllocateEx routine to allocate storage for a new entry, if the driver has provided such a routine. Otherwise, a default allocation routine is used to allocate the entry.

After the caller finishes using the entry, it should free the entry by calling the ExFreeToLookasideListEx routine.

In the current implementation, a lookaside list operates as a last-in, first-out (LIFO) stack. Thus, the last entry to be freed (and pushed onto the stack) is the next entry to be allocated (popped) from the list.

For more information about lookaside lists, see Using Lookaside Lists.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExFreeToLookasideListEx

ExInitializeLookasideListEx

LOOKASIDE_LIST_EX

LookasideListAllocateEx

ExAllocateFromNPagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAllocateFromNPagedLookasideList** routine returns a pointer to a nonpaged entry from the given lookaside list, or it returns a pointer to a newly allocated nonpaged entry.

Syntax

```
PVOID ExAllocateFromNPagedLookasideList(
  PNPAGED_LOOKASIDE_LIST Lookaside
);
```

Parameters

`Lookaside`

A pointer to the NPAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller already initialized with ExInitializeNPagedLookasideList.

Return Value

ExAllocateFromNPagedLookasideList returns a pointer to an entry if one can be allocated. Otherwise, it returns **NULL**.

Remarks

If the given lookaside list is not empty, **ExAllocateFromNPagedLookasideList** removes the first entry from the list and returns a pointer to this entry. Otherwise, **ExAllocateFromNPagedLookasideList** either calls the *Allocate* routine specified at list initialization or ExAllocatePoolWithTag to return an entry pointer.

The caller can then set up the returned entry with any caller-determined data. For example, a driver might use each such fixed-size entry to set up command blocks, like SCSI SRBs, to peripheral devices on a particular type of I/O bus. The caller should release each entry with ExFreeToNPagedLookasideList when it is no longer in use.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL"

See Also

ExAllocateFromPagedLookasideList

ExFreeToNPagedLookasideList

ExInitializeNPagedLookasideList

NPAGED_LOOKASIDE_LIST

ExAllocateFromPagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAllocateFromPagedLookasideList** routine returns a pointer to a paged entry from the given lookaside list, or it returns a pointer to a newly allocated paged entry.

Syntax

```
PVOID ExAllocateFromPagedLookasideList(
  PPAGED_LOOKASIDE_LIST Lookaside
);
```

Parameters

`Lookaside`

A pointer to the PAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller already initialized with ExInitializePagedLookasideList.

Return Value

ExAllocateFromPagedLookasideList returns a pointer to an entry if one can be allocated. Otherwise, it returns **NULL**.

Remarks

If the given lookaside list is not empty, **ExAllocateFromPagedLookasideList** removes the first entry from the list and returns a pointer to this entry. Otherwise, **ExAllocateFromPagedLookasideList** either calls the *Allocate* routine specified at list initialization or ExAllocatePoolWithTag to return an entry pointer.

The caller can then set up the returned entry with any caller-determined data. The caller should release each entry with ExFreeToPagedLookasideList when it is no longer in use.

Because the entries in a paged lookaside list are allocated from pageable memory, they must not be accessed at an IRQL >= DISPATCH_LEVEL. You can use ExAllocateFromNPagedLookasideList to create a lookaside list with non-pageable entries.

On Windows 2000, drivers must use the **-D_WIN2K_COMPAT_SLIST_USAGE** switch to successfully link code that uses **ExAllocateFromPagedLookasideList**.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

ExAllocateFromNPagedLookasideList

ExFreeToPagedLookasideList

ExInitializePagedLookasideList

PAGED_LOOKASIDE_LIST

ExAllocatePool function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAllocatePool** routine is obsolete, and is exported only for existing binaries. Use ExAllocatePoolWithTag instead.

ExAllocatePool allocates pool memory of the specified type and returns a pointer to the allocated block.

Syntax

```
PVOID ExAllocatePool(
  __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  SIZE_T                                         NumberOfBytes
);
```

Parameters

`PoolType`

Specifies the type of pool memory to allocate. For a description of the available pool memory types, see POOL_TYPE.

You can modify *PoolType* by using a bitwise OR with the POOL_COLD_ALLOCATION flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_COLD_ALLOCATION flag is only advisory and is available for Windows XP and later versions of the Windows operating system.

`NumberOfBytes`

Specifies the number of bytes to allocate.

Return Value

ExAllocatePool returns **NULL** if there is insufficient memory in the free pool to satisfy the request. Otherwise the routine returns a pointer to the allocated memory.

Remarks

This routine is used for the general pool allocation of memory.

If *NumberOfBytes* is PAGE_SIZE or greater, a page-aligned buffer is allocated. Memory allocations of PAGE_SIZE or less do not cross page boundaries. Memory allocations of less than PAGE_SIZE are not necessarily page-aligned but are aligned to 8-byte boundaries in 32-bit systems and to 16-byte boundaries in 64-bit systems.

A successful allocation requesting *NumberOfBytes* < PAGE_SIZE of nonpaged pool gives the caller exactly the number of requested bytes of memory. If an allocation request for *NumberOfBytes* > PAGE_SIZE succeeds and *NumberOfBytes* is not an exact multiple of PAGE_SIZE, the last page in the allocation contains bytes that are not part of the caller's allocation. If possible, the pool allocator uses these bytes. To avoid corrupting data that belongs to other kernel-mode components, drivers must access only storage addresses that they have explicitly allocated.

If **ExAllocatePool** returns **NULL**, the caller should return the NTSTATUS value

STATUS_INSUFFICIENT_RESOURCES or should delay processing to another point in time.

Callers of **ExAllocatePool** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the page type.

> **Note** Do not set *NumberOfBytes* = 0. Avoid zero-length allocations because they waste pool header space and, in many cases, indicate a potential validation issue in the calling code. For this reason, Driver Verifier flags such allocations as possible errors.

The system automatically sets certain standard event objects when the amount of pool (paged or nonpaged) is high or low. Drivers can wait for these events to tune their pool usage. For more information, see Standard Event Objects.

> **Note** Memory that **ExAllocatePool** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Obsolete. This routine is exported only for existing binaries. Use ExAllocatePoolWithTag instead.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Classpnp.h, Smcnt.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExAllocatePoolWithTag

ExFreePool

POOL_TYPE

ExAllocatePoolWithQuota function

1/25/2019 • 2 minutes to read • Edit Online

The **ExAllocatePoolWithQuota** routine is obsolete, and is exported only for existing driver binaries. Use ExAllocatePoolWithQuotaTag instead.

ExAllocatePoolWithQuota allocates pool memory, charging quota against the current process.

Syntax

```
PVOID ExAllocatePoolWithQuota(
  __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  SIZE_T                                         NumberOfBytes
);
```

Parameters

`PoolType`

Specifies the type of pool memory to allocate. For a description of the available pool memory types, see POOL_TYPE.

You can modify *PoolType* by using a bitwise OR with the POOL_COLD_ALLOCATION flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_COLD_ALLOCATION flag is only advisory and is available for Windows XP and later versions of the Windows operating system.

`NumberOfBytes`

Specifies the number of bytes to allocate.

Return Value

ExAllocatePoolWithQuota returns a pointer to the allocated pool.

If the request cannot be satisfied, **ExAllocatePoolWithQuota** raises an exception.

Remarks

This routine is called by highest-level drivers that allocate memory to satisfy a request in the context of the process that originally made the I/O request. Lower-level drivers call ExAllocatePoolWithTag instead.

If *NumberOfBytes* is PAGE_SIZE or greater, a page-aligned buffer is allocated. Quota is not charged to the process for allocations of PAGE_SIZE or greater.

Memory allocations of less than PAGE_SIZE are allocated within a page and do not cross page boundaries. Memory allocations of PAGE_SIZE or less are not necessarily page-aligned but are aligned to 8-byte boundaries in 32-bit systems and to 16-byte boundaries in 64-bit systems.

Note Do not set *NumberOfBytes* = 0. Avoid zero-length allocations because they waste pool header space and, in many cases, indicate a potential validation issue in the calling code. For this reason, Driver Verifier flags such allocations as possible errors.

The system automatically sets certain standard event objects when the amount of pool (paged or nonpaged) is high or low. Drivers can wait for these events to tune their pool usage. For more information, see Standard Event Objects.

Note Memory that **ExAllocatePoolWithQuota** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Callers of **ExAllocatePoolWithQuota** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the pool type.

Requirements

Minimum supported client	Obsolete. This routine is exported only for existing driver binaries. Use ExAllocatePoolWithQuotaTag instead.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Classpnp.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExFreePool

POOL_TYPE

ExAllocatePoolWithQuotaTag function

1/25/2019 • 3 minutes to read • Edit Online

The **ExAllocatePoolWithQuotaTag** routine allocates pool memory, charging the quota against the current process.

Syntax

```
PVOID ExAllocatePoolWithQuotaTag(
  __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  SIZE_T                                         NumberOfBytes,
  ULONG                                          Tag
);
```

Parameters

`PoolType`

Specifies the type of pool memory to allocate. For a description of the available pool memory types, see POOL_TYPE.

You can modify the *PoolType* value by bitwise-ORing this value with the POOL_QUOTA_FAIL_INSTEAD_OF_RAISE flag. This flag causes the routine to return a **NULL** value if the request cannot be satisfied.

Similarly, you can modify the *PoolType* value by bitwise-ORing this value with the POOL_COLD_ALLOCATION flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_COLD_ALLOCATION flag is only advisory and is supported in Windows XP and later versions of the Windows operating system.

`NumberOfBytes`

Specifies the number of bytes to allocate.

`Tag`

Specifies the pool tag for the allocated memory. For more information, see the *Tag* parameter of ExAllocatePoolWithTag.

Return Value

ExAllocatePoolWithQuotaTag returns a pointer to the allocated pool.

If the request cannot be satisfied, **ExAllocatePoolWithQuotaTag** raises an exception unless POOL_QUOTA_FAIL_INSTEAD_OF_RAISE is specified. Using POOL_QUOTA_FAIL_INSTEAD_OF_RAISE is preferred for performance reasons.

Remarks

This routine is called by highest-level drivers that allocate memory to satisfy a request in the context of the process that originally made the I/O request. Lower-level drivers call ExAllocatePoolWithTag instead.

If *NumberOfBytes* is PAGE_SIZE or greater, a page-aligned buffer is allocated. Memory allocations of PAGE_SIZE or less are allocated within a page and do not cross page boundaries. Memory allocations of less than PAGE_SIZE are not necessarily page-aligned but are aligned to 8-byte boundaries in 32-bit systems and to 16-byte boundaries in 64-bit systems.

The system associates the pool tag with the allocated memory. Programming tools, such as WinDbg, can display the pool tag associated with each allocated buffer. The value of *Tag* is normally displayed in reversed order. For example, if a caller passes 'Fred' as a *Tag*, it would appear as 'derF' if the pool is dumped or when tracking pool usage in the debugger.

The allocated buffer can be freed with either ExFreePool or ExFreePoolWithTag.

> **Note** Do not set *NumberOfBytes* = 0. Avoid zero-length allocations because they waste pool header space and, in many cases, indicate a potential validation issue in the calling code. For this reason, Driver Verifier flags such allocations as possible errors.

The system automatically sets certain standard event objects when the amount of pool (paged or nonpaged) is high or low. Drivers can wait for these events to tune their pool usage. For more information, see Standard Event Objects.

In a non-uniform memory access (NUMA) multiprocessor architecture, **ExAllocatePoolWithQuotaTag** tries to allocate memory that is local to the processor that is calling **ExAllocatePoolWithQuotaTag**. If no local memory is available, **ExAllocatePoolWithQuotaTag** allocates the closest available memory.

> **Note** Memory that **ExAllocatePoolWithQuotaTag** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Callers of **ExAllocatePoolWithQuotaTag** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the pool type.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExAllocatePoolWithTag

ExFreePool

ExFreePoolWithTag

POOL_TYPE

ExAllocatePoolWithTag function

1/25/2019 • 3 minutes to read • Edit Online

The **ExAllocatePoolWithTag** routine allocates pool memory of the specified type and returns a pointer to the allocated block.

Syntax

```
PVOID ExAllocatePoolWithTag(
  __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  SIZE_T                                         NumberOfBytes,
  ULONG                                          Tag
);
```

Parameters

`PoolType`

The type of pool memory to allocate. For a description of the available pool memory types, see POOL_TYPE.

You can modify the *PoolType* value by bitwise-ORing this value with the POOL_RAISE_IF_ALLOCATION_FAILURE flag. This flag causes an exception to be raised if the request cannot be satisfied. Use of the POOL_RAISE_IF_ALLOCATION_FAILURE flag is not recommended because it is costly.

Similarly, you can modify the *PoolType* value by bitwise-ORing this value with the POOL_COLD_ALLOCATION flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_COLD_ALLOCATION flag is only advisory and is supported starting with Windows XP.

`NumberOfBytes`

The number of bytes to allocate.

`Tag`

The pool tag to use for the allocated memory. Specify the pool tag as a character literal of up to four characters delimited by single quotation marks (for example, 'Tag1'). The string is usually specified in reverse order (for example, '1gaT'). Each ASCII character in the tag must be a value in the range 0x20 (space) to 0x7E (tilde). Each allocation code path should use a unique pool tag to help debuggers and verifiers identify the code path.

Return Value

ExAllocatePoolWithTag returns **NULL** if there is insufficient memory in the free pool to satisfy the request. Otherwise, the routine returns a pointer to the allocated memory.

Remarks

This routine is used for the general pool allocation of memory.

If *NumberOfBytes* is PAGE_SIZE or greater, a page-aligned buffer is allocated. Memory allocations of PAGE_SIZE or less are allocated within a page and do not cross page boundaries. Memory allocations of less than PAGE_SIZE are not necessarily page-aligned but are aligned to 8-byte boundaries in 32-bit systems and to 16-byte boundaries

in 64-bit systems.

A successful allocation requesting *NumberOfBytes* < PAGE_SIZE of nonpaged pool gives the caller exactly the number of requested bytes of memory. If an allocation request for *NumberOfBytes* > PAGE_SIZE succeeds and *NumberOfBytes* is not an exact multiple of PAGE_SIZE, the last page in the allocation contains bytes that are not part of the caller's allocation. If possible, the pool allocator uses these bytes. To avoid corrupting data that belongs to other kernel-mode components, drivers must access only storage addresses that they have explicitly allocated.

The system associates the pool tag with the allocated memory. Programming tools, such as WinDbg, can display the pool tag associated with each allocated buffer. Gflags, a tool included in Debugging Tools for Windows, turns on a system feature that requests allocation from special pool for a particular pool tag. Poolmon, which is included in the WDK, tracks memory by pool tag.

The value of *Tag* is stored, and sometimes displayed, in reverse (little-endian) order. For example, if a caller passes 'Fred' as a *Tag*, it appears as "derF" in a pool dump and in pool usage tracking in the debugger, and as 0x64657246 in the registry and in tool displays.

The allocated buffer can be freed with either ExFreePool or ExFreePoolWithTag.

The system automatically sets certain standard event objects when the amount of pool (paged or nonpaged) is high or low. Drivers can wait for these events to tune their pool usage. For more information, see Standard Event Objects.

Callers of **ExAllocatePoolWithTag** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the page type.

> **Note** Do not set *NumberOfBytes* = 0. Avoid zero-length allocations because they waste pool header space and, in many cases, indicate a potential validation issue in the calling code. For this reason, Driver Verifier flags such allocations as possible errors.

In a non-uniform memory access (NUMA) multiprocessor architecture, **ExAllocatePoolWithTag** tries to allocate memory that is local to the processor that is calling **ExAllocatePoolWithTag**. If no local memory is available, **ExAllocatePoolWithTag** allocates the closest available memory.

> **Note** Memory that **ExAllocatePoolWithTag** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

DDI compliance rules	CheckDeviceObjectFlags, IrqlExAllocatePool, IrqlExFree1, PowerDownAllocate, PowerUpFail, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTagPriority

ExFreePool

ExFreePoolWithTag

POOL_TYPE

ExAllocatePoolWithTagPriority function

1/25/2019 • 3 minutes to read • Edit Online

The **ExAllocatePoolWithTagPriority** routine allocates pool memory of the specified type.

Syntax

```
PVOID ExAllocatePoolWithTagPriority(
  __drv_strictTypeMatch(__drv_typeCond)POOL_TYPE        PoolType,
  SIZE_T                                                NumberOfBytes,
  ULONG                                                 Tag,
  __drv_strictTypeMatch(__drv_typeExpr)EX_POOL_PRIORITY Priority
);
```

Parameters

`PoolType`

The type of pool memory to allocate. For a description of the available pool memory types, see POOL_TYPE.

You can modify the *PoolType* value by bitwise-ORing this value with the POOL_RAISE_IF_ALLOCATION_FAILURE flag. This flag causes an exception to be raised if the request cannot be satisfied.

Similarly, you can modify the *PoolType* value by bitwise-ORing this value with the POOL_COLD_ALLOCATION flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_COLD_ALLOCATION flag is only advisory and is available starting with Windows XP.

`NumberOfBytes`

The number of bytes to allocate.

`Tag`

The pool tag to use for the allocated memory. For more information, see the *Tag* parameter of ExAllocatePoolWithTag.

`Priority`

The priority of this request. Set this parameter to one of the following **EX_POOL_PRIORITY** enumeration values.

PRIORITY VALUE	DESCRIPTION
LowPoolPriority	Specifies that the system may fail the request when it runs low on resources. Driver allocations that can recover from an allocation failure use this priority.
NormalPoolPriority	Specifies that the system may fail the request when it runs very low on resources. Most drivers should use this value.

HighPoolPriority	Specifies that the system must not fail the request, unless it is completely out of resources. Drivers only use this value when it is critically important for the request to succeed.

The **EX_POOL_PRIORITY** enumeration defines ***Xxx*SpecialPoolOverrun** and ***Xxx*SpecialPoolUnderrun** variants to specify how memory should be allocated when Driver Verifier (or special pool) is enabled. If the driver specifies ***Xxx*SpecialPoolUnderrun**, when the memory manager allocates memory from special pool, it allocates it at the start of a physical page. If the driver specifies ***Xxx*SpecialPoolOverrun**, the memory manager allocates it at the end of a physical page.

Return Value

ExAllocatePoolWithTagPriority returns **NULL** if there is insufficient memory in the free pool to satisfy the request unless POOL_RAISE_IF_ALLOCATION_FAILURE is specified. Otherwise, the routine returns a pointer to the allocated memory.

Remarks

Callers of **ExAllocatePoolWithTagPriority** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the page type.

If *NumberOfBytes* is PAGE_SIZE or greater, a page-aligned buffer is allocated. Memory allocations of PAGE_SIZE or less are allocated within a page and do not cross page boundaries. Memory allocations of less than PAGE_SIZE are not necessarily page-aligned but are aligned to 8-byte boundaries in 32-bit systems and to 16-byte boundaries in 64-bit systems.

> **Note** Do not set *NumberOfBytes* = 0. Avoid zero-length allocations because they waste pool header space and, in many cases, indicate a potential validation issue in the calling code. For this reason, Driver Verifier flags such allocations as possible errors.

The system automatically sets certain standard event objects when the amount of pool (paged or nonpaged) is high or low. Drivers can wait for these events to tune their pool usage. For more information, see Standard Event Objects. In a non-uniform memory access (NUMA) multiprocessor architecture, **ExAllocatePoolWithTagPriority** tries to allocate memory that is local to the processor that is calling **ExAllocatePoolWithTagPriority**. If no local memory is available, **ExAllocatePoolWithTagPriority** allocates the closest available memory.

> **Note** Memory that **ExAllocatePoolWithTagPriority** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IrqlExAllocatePool, IrqlExFree2, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExAllocatePoolWithTag

ExFreePool

POOL_TYPE

ExAllocateTimer function

1/25/2019 • 3 minutes to read • Edit Online

The **ExAllocateTimer** routine allocates and initializes a timer object.

Syntax

```
PEX_TIMER ExAllocateTimer(
  PEXT_CALLBACK Callback,
  PVOID         CallbackContext,
  ULONG         Attributes
);
```

Parameters

`Callback`

A pointer to a driver-implemented ExTimerCallback callback routine. The operating system calls this routine when the timer expires. This parameter is optional and can be **NULL** if no callback routine is needed.

`CallbackContext`

A context value for the callback routine pointed to by the *Callback* parameter. The operating system passes this value as a parameter to the *ExTimerCallback* callback routine, if one is specified. This parameter is typically a pointer to a caller-defined structure that contains context information used by the callback routine. This parameter is optional and can be set to **NULL** if no context information is needed.

`Attributes`

The timer attributes. Set this parameter to zero or to the bitwise-OR of one or more of the following timer flag bits.

TIMER FLAG BIT	DESCRIPTION
EX_TIMER_HIGH_RESOLUTION	High-resolution timer. Make the timer more precise by using a higher-resolution clock to drive the timer.
EX_TIMER_NO_WAKE	No-wake timer. Make the timer delay waking the processor to expire until the timer's expiration time plus its delay tolerance is exceeded.
EX_TIMER_NOTIFICATION	Notification timer. Make the timer a notification timer instead of a synchronization timer. If this flag is not set, the timer is a synchronization timer.

The EX_TIMER_NOTIFICATION flag bit can be set regardless of what other flag bits are set.

The EX_TIMER_HIGH_RESOLUTION and EX_TIMER_NO_WAKE flag bits are mutually exclusive. If the caller sets both of these flag bits, the routine bug checks.

For more information about timer attributes, see Remarks.

Return Value

This routine returns a pointer to an EX_TIMER structure, if the call is successful. This structure is the timer object that the routine has allocated and initialized. If the call fails, the routine returns **NULL**.

Remarks

This routine returns a pointer to the new timer object. To use the timer, the calling driver supplies this pointer in subsequent calls to the ExSetTimer, ExCancelTimer, and ExDeleteTimer routines. If the driver supplies a pointer to an ExTimerCallback callback routine as an input parameter to the **ExAllocateTimer** routine, the operating system passes this timer object as an input parameter to the *ExTimerCallback* routine.

A timer can be a notification timer or a synchronization timer. When a notification timer is signaled, all waiting threads have their wait satisfied. The state of this timer remains signaled until it is explicitly reset. When a synchronization timer expires, its state is set to signaled until a single waiting thread is released. Then, the timer is reset to the not-signaled state.

If the EX_TIMER_HIGH_RESOLUTION flag bit is set in *Attributes*, the operating system increases the resolution of the system clock, as necessary, so that the times at which the timer expires more precisely corresponds to the nominal expiration times specified in the *DueTime* and *Period* parameters to the **ExSetTimer** routine. For more information, see High-Resolution Timers.

If the EX_TIMER_NO_WAKE flag bit is set in *Attributes*, the timer avoids unnecessarily waking the processor from a low-power state. For more information, see No-Wake Timers.

ExAllocateTimer allocates the storage for the timer object. When this object is no longer needed, the caller is responsible for freeing this object by calling the **ExDeleteTimer** routine.

The *Callback* parameter is optional. A driver that does not supply an *ExTimerCallback* routine can instead initiate a wait operation on the timer object. A driver thread can call a routine such as KeWaitForSingleObject or KeWaitForMultipleObjects to wait for the timer to expire. When the timer expires, the timer object is signaled.

For more information, see ExXxxTimer Routines and EX_TIMER Objects.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
IRQL	"<= DISPATCH_LEVEL"

See Also

EX_TIMER

ExCancelTimer

ExDeleteTimer

ExSetTimer

ExTimerCallback

ExCancelTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **ExCancelTimer** routine cancels a timer that was set by a previous call to the ExSetTimer routine.

Syntax

```
BOOLEAN ExCancelTimer(
  PEX_TIMER              Timer,
  PEXT_CANCEL_PARAMETERS Parameters
);
```

Parameters

`Timer`

A pointer to an EX_TIMER structure. This structure is a timer object that was previously allocated by the ExAllocateTimer routine.

`Parameters`

Drivers must set this parameter to **NULL**.

Return Value

This routine returns **TRUE** if the timer was canceled. Otherwise, the routine returns **FALSE**.

Remarks

After your driver calls the ExSetTimer routine to set the timer, the timer might be canceled before it expires. The driver can call the **ExCancelTimer** routine to explicitly cancel a pending timer. Also, if the driver calls the **ExSetTimer** routine to set a new timer on a timer object before a previously set timer on this object expires, this **ExSetTimer** call implicitly cancels the previous timer and then starts the new timer.

A return value of **TRUE** indicates that the timer was set when **ExCancelTimer** was called. In this case, the routine cancels the timer. A return value of **FALSE** indicates that the timer was not set, which might mean that the timer was never set, that the timer had already been canceled, or that the timer had already expired.

When a timer expires, the operating system signals the timer object. As an option, a driver can supply an ExTimerCallback callback routine that is called when the timer expires. If the driver supplies this callback routine, this routine is guaranteed to be called if the timer object is signaled. If the timer is canceled before it expires, the timer object is not signaled and the callback routine is not called.

For more information, see ExXxxTimer Routines and EX_TIMER Objects.

Requirements

Minimum supported client	Available starting with Windows 8.1.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
IRQL	"<= DISPATCH_LEVEL"

See Also

EX_TIMER

ExAllocateTimer

ExSetTimer

ExConvertExclusiveToSharedLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExConvertExclusiveToSharedLite** routine converts a given resource from acquired for exclusive access to acquired for shared access.

Syntax

```
void ExConvertExclusiveToSharedLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the resource for which the access should be converted.

Return Value

None

Remarks

The caller must have exclusive access to the given resource. During this conversion, the current thread and any other threads waiting for shared access to the resource are given shared access.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte3, HwStorPortProhibitedDDIs

See Also

ExIsResourceAcquiredExclusiveLite

ExCreateCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **ExCreateCallback** routine either creates a new callback object or opens an existing callback object on behalf of the caller.

Syntax

```
NTSTATUS ExCreateCallback(
  PCALLBACK_OBJECT   *CallbackObject,
  POBJECT_ATTRIBUTES ObjectAttributes,
  BOOLEAN            Create,
  BOOLEAN            AllowMultipleCallbacks
);
```

Parameters

`CallbackObject`

A pointer to a location that receives a pointer to a callback object, which is an opaque, system structure. If the **ExCreateCallback** call succeeds, the routine writes the address of the newly created or opened callback object to this location. The callback object pointer obtained from this routine can be supplied as a parameter to the ExRegisterCallback or ExNotifyCallback routine.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that contains the callback object's attributes. This structure was previously initialized by the InitializeObjectAttributes routine.

`Create`

Whether to create a callback object. Set to **TRUE** to create a new callback object if the requested object cannot be opened. Otherwise, set to **FALSE**.

`AllowMultipleCallbacks`

Whether a newly created callback object should allow multiple registered callback routines. Set to **TRUE** to allow multiple registered callback routines. Otherwise, set to **FALSE**. This parameter is ignored when *Create* is **FALSE** or when opening an existing object.

Return Value

ExCreateCallback returns STATUS_SUCCESS if a callback object was opened or created. Otherwise, it returns an NTSTATUS error code to indicate the nature of the failure.

Remarks

A driver calls **ExCreateCallback** to create a new callback object or to open an existing callback object. After the object has been created or opened, other components can call the ExRegisterCallback routine to register callback routines with the callback object.

Before calling **ExCreateCallback**, the driver must call InitializeObjectAttributes to initialize the

OBJECT_ATTRIBUTES structure for the callback object. Unnamed callback objects are not permitted. The caller must specify a name for the object; otherwise, the call fails with STATUS_UNSUCCESSFUL. The caller should specify the OBJ_PERMANENT attribute in the callback object to prevent the object from being deleted before it can be registered with the object manager. The caller should also specify any other attributes, such as OBJ_CASE_INSENSITIVE, that might be needed.

When all operations have been completed with the callback object, the driver must delete the object to prevent a memory leak. For information about deleting an object that was created with the OBJ_PERMANENT object attribute, see ObDereferenceObject.

The following table shows the callback objects that the operating system creates for use by drivers.

CALLBACK OBJECT NAME	USAGE
\Callback\SetSystemTime	The operating system calls any callback routines registered for this object whenever the system time changes.
\Callback\PowerState	The operating system calls any callback routines registered for this object whenever certain system power characteristics change. When a driver registers for callback notification (by calling **ExRegisterCallback**), it can specify the changes for which it should be notified.

For more information about callback objects, see Callback Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExPassive, HwStorPortProhibitedDDIs

See Also

ExNotifyCallback

ExRegisterCallback

InitializeObjectAttributes

ObDereferenceObject

ExDeleteLookasideListEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExDeleteLookasideListEx** routine deletes a lookaside list.

Syntax

```
void ExDeleteLookasideListEx(
  PLOOKASIDE_LIST_EX Lookaside
);
```

Parameters

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes a lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

Return Value

None

Remarks

ExDeleteLookasideListEx is the reciprocal of the **ExInitializeLookasideListEx** routine. It frees any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

On return from **ExDeleteLookasideListEx**, the caller is responsible for releasing, if necessary, the caller-supplied storage for the **LOOKASIDE_LIST_EX** structure.

For more information about lookaside lists, see Using Lookaside Lists.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExInitializeLookasideListEx

LOOKASIDE_LIST_EX

ExDeleteNPagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExDeleteNPagedLookasideList** routine destroys a nonpaged lookaside list.

Syntax

```
void ExDeleteNPagedLookasideList(
  PNPAGED_LOOKASIDE_LIST Lookaside
);
```

Parameters

`Lookaside`

A pointer to the NPAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller originally set up with ExInitializeNPagedLookasideList.

Return Value

None

Remarks

ExDeleteNPagedLookasideList is the reciprocal of **ExInitializeNPagedLookasideList**. It frees any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

The caller of **ExDeleteNPagedLookasideList** is responsible for subsequently releasing the memory that the caller provided to contain the list head.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExInitializeNPagedLookasideList

NPAGED_LOOKASIDE_LIST

ExDeletePagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExDeletePagedLookasideList** routine destroys a paged lookaside list.

Syntax

```
void ExDeletePagedLookasideList(
  PPAGED_LOOKASIDE_LIST Lookaside
);
```

Parameters

`Lookaside`

A pointer to the PAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller already initialized with ExInitializePagedLookasideList, which the caller originally set up with **ExInitializePagedLookasideList**.

Return Value

None

Remarks

ExDeletePagedLookasideList is the reciprocal of **ExInitializePagedLookasideList**. It frees any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

The caller of **ExDeletePagedLookasideList** is responsible for subsequently releasing the memory that the caller provided to contain the list head.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ExInitializePagedLookasideList

PAGED_LOOKASIDE_LIST

ExDeleteResourceLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExDeleteResourceLite** routine deletes a given resource from the system's resource list.

Syntax

```
NTSTATUS ExDeleteResourceLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the caller-supplied storage for the initialized resource variable to be deleted.

Return Value

ExDeleteResourceLite returns STATUS_SUCCESS if the resource was deleted.

Remarks

After calling **ExDeleteResourceLite**, the caller can free the memory it allocated for its resource.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte3, HwStorPortProhibitedDDIs

See Also

ExFreePool

ExInitializeResourceLite

ExReinitializeResourceLite

ExDeleteTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **ExDeleteTimer** routine deletes a timer object that was previously allocated by the ExAllocateTimer routine.

Syntax

```
BOOLEAN ExDeleteTimer(
  PEX_TIMER              Timer,
  BOOLEAN                Cancel,
  BOOLEAN                Wait,
  PEXT_DELETE_PARAMETERS Parameters
);
```

Parameters

`Timer`

A pointer to an EX_TIMER structure. This structure is a timer object that was previously allocated by the ExAllocateTimer routine.

`Cancel`

Whether to cancel the timer if it is currently set. If **TRUE**, the routine cancels the timer if the timer is set. If **FALSE**, the routine does not cancel the timer if it is set. If the caller sets *Wait* to **TRUE**, *Cancel* must also be **TRUE**; otherwise, a bug check occurs.

`Wait`

Whether to wait for the timer to finish expiring if the timer has already started the process of expiring. If **TRUE**, the routine waits for the timer to expire. If **FALSE**, the routine does not wait. For more information, see Remarks.

`Parameters`

A pointer to an EXT_DELETE_PARAMETERS structure. The calling driver previously called the ExInitializeDeleteTimerParameters routine to initialize this structure.

Return Value

This routine returns **TRUE** if *Cancel* is **TRUE** and the timer was canceled. Otherwise, the routine returns **FALSE**. For more information, see Remarks.

Remarks

After your driver calls this routine, the timer object pointed to by *Timer* might no longer be valid. However, the *Timer* parameter value passed to an ExTimerCallback callback routine, if the driver implements this routine, is always a valid pointer to a timer object.

If *Cancel* is **TRUE**, a return value of **FALSE** indicates that the timer was never set, or that the timer was set but expired before it could be canceled. If *Cancel* is **FALSE**, the routine always returns **FALSE**.

This routine can block if *Wait* is **TRUE**. If **ExDeleteTimer** is called from the driver's *ExTimerCallback* routine, which is called at DISPATCH_LEVEL, the *Wait* parameter in this call must be **FALSE**.

If *Wait* is **TRUE**, the routine must be called at IRQL <= APC_LEVEL. If *Wait* is **FALSE**, the routine can be called at IRQL <= DISPATCH_LEVEL.

For more information, see Deleting a System-Allocated Timer Object.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
IRQL	See Remarks.

See Also

EXT_DELETE_PARAMETERS

EX_TIMER

ExAllocateTimer

ExInitializeDeleteTimerParameters

ExTimerCallback

ExTimerDeleteCallback

ExFlushLookasideListEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFlushLookasideListEx** routine flushes all entries from the specified lookaside list and frees the allocated storage for each entry.

Syntax

```
void ExFlushLookasideListEx(
  PLOOKASIDE_LIST_EX Lookaside
);
```

Parameters

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes a lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

Return Value

None

Remarks

On return from this routine, the lookaside list is empty and the allocated storage for all its entries has been freed.

For more information about lookaside lists, see Using Lookaside Lists.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExInitializeLookasideListEx

ExFreePool function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreePool** routine deallocates a block of pool memory.

Syntax

```
void ExFreePool(
  PVOID P
);
```

Parameters

P

Return Value

None

Remarks

This routine releases memory allocated by ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePoolWithQuota, or ExAllocatePoolWithQuotaTag. The memory block must not be accessed after it is freed.

Drivers can also use the **ExFreePoolWithTag** routine to free buffers allocated by ExAllocatePoolWithTag and ExAllocatePoolWithQuotaTag.

Callers of **ExFreePool** must be running at IRQL <= DISPATCH_LEVEL. A caller at DISPATCH_LEVEL must have specified a **NonPaged***XxxPoolType* when the memory was allocated. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	DoubleExFreePool

See Also

ExAllocatePool

ExAllocatePoolWithQuota

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExFreePoolWithTag

ExFreePoolWithTag function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreePoolWithTag** routine deallocates a block of pool memory allocated with the specified tag.

Syntax

```
void ExFreePoolWithTag(
  PVOID P,
  ULONG Tag
);
```

Parameters

`P`

Specifies the beginning address of a block of pool memory allocated by either ExAllocatePoolWithTag or ExAllocatePoolWithQuotaTag.

`Tag`

Specifies the tag value passed to ExAllocatePoolWithTag or ExAllocatePoolWithQuotaTag when the block of memory was originally allocated.

NOTE

Tag must be a value from 0x20 (space) to 0x126 (tilde).

Return Value

None

Remarks

Callers of **ExFreePoolWithTag** must be running at IRQL <= DISPATCH_LEVEL. A caller at DISPATCH_LEVEL must have specified a **NonPaged***XxxPoolType* when the memory was allocated. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IrqlExFree1, IrqlExFree2, IrqlExFree3

See Also

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExFreePool

ExFreeToLookasideListEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreeToLookasideListEx** routine inserts an entry into a lookaside list, or, if the list is full, frees the allocated storage for the entry.

Syntax

```
void ExFreeToLookasideListEx(
  PLOOKASIDE_LIST_EX Lookaside,
  PVOID              Entry
);
```

Parameters

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes a lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

`Entry`

A pointer to the lookaside-list entry that is being freed.

Return Value

None

Remarks

This routine frees a lookaside-list entry that was allocated by a previous call to the ExAllocateFromLookasideListEx routine. **ExFreeToLookasideListEx** inserts the entry into the specified lookaside list, if space for the entry is available in the list. If the list is full (that is, it already contains the maximum number of entries, as determined by the operating system), **ExFreeToLookasideListEx** calls the LookasideListFreeEx routine to free the storage for the specified entry, if the driver has provided such a routine. Otherwise, a default deallocation routine is used to free the entry.

For more information about lookaside lists, see Using Lookaside Lists.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExAllocateFromLookasideListEx

ExInitializeLookasideListEx

LOOKASIDE_LIST_EX

ExFreeToNPagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreeToNPagedLookasideList** routine returns a nonpaged entry to the given lookaside list or to nonpaged pool.

Syntax

```
void ExFreeToNPagedLookasideList(
  PNPAGED_LOOKASIDE_LIST Lookaside,
  PVOID                  Entry
);
```

Parameters

`Lookaside`

A pointer to the NPAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller already initialized with ExInitializeNPagedLookasideList.

`Entry`

A pointer to the entry to be freed. The caller obtained this pointer from a preceding call to ExAllocateFromNPagedLookasideList.

Return Value

None

Remarks

ExFreeToNPagedLookasideList is the reciprocal of **ExAllocateFromNPagedLookasideList**. It releases a caller-allocated entry back to the caller's lookaside list or to nonpaged pool when that entry is no longer in use.

The same entry can be reallocated or another entry allocated later with a subsequent call to **ExAllocateFromNPagedLookasideList**. The user of the lookaside list can allocate and free such entries dynamically on an as-needed basis until it calls **ExDeleteNPagedLookasideList**, which releases any outstanding entries in the list before it clears the system state for the given lookaside list and returns control.

If the specified lookaside list has not yet reached the system-determined maximum number of entries, **ExFreeToNPagedLookasideList** inserts the given entry at the front of the list. Otherwise, the buffer at *Entry* is released to nonpaged pool using the caller-supplied *Free* routine, if any, that was set up when the lookaside list was initialized or **ExFreePool**.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	SpIrql

See Also

ExAllocateFromNPagedLookasideList

ExDeleteNPagedLookasideList

ExInitializeNPagedLookasideList

NPAGED_LOOKASIDE_LIST

ExFreeToPagedLookasideList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExFreeToPagedLookasideList** routine returns a pageable entry to the given lookaside list or to paged pool.

Syntax

```
void ExFreeToPagedLookasideList(
  PPAGED_LOOKASIDE_LIST Lookaside,
  PVOID                 Entry
);
```

Parameters

`Lookaside`

A pointer to the PAGED_LOOKASIDE_LIST structure for the lookaside list, which the caller already initialized with ExInitializePagedLookasideList, which the caller already initialized with **ExInitializePagedLookasideList**.

`Entry`

A pointer to the entry to be freed. The caller obtained this pointer from a preceding call to ExAllocateFromPagedLookasideList.

Return Value

None

Remarks

ExFreeToPagedLookasideList is the reciprocal of **ExAllocateFromPagedLookasideList**. It releases a caller-allocated entry back to the caller's lookaside list or to paged pool when that entry is no longer in use.

The same entry can be reallocated or another entry can be allocated later with a subsequent call to **ExAllocateFromPagedLookasideList**. The user of a lookaside list can allocate and free such entries dynamically, as needed, until it calls **ExDeletePagedLookasideList**. **ExDeletePagedLookasideList** releases any outstanding entries in the list before it clears the system state for the given lookaside list and returns control.

If the specified lookaside list has not yet reached the system-determined maximum number of entries, **ExFreeToPagedLookasideList** inserts the given entry at the front of the list. Otherwise, the buffer at *Entry* is released back to paged pool using the caller-supplied *Free* routine, if any, that was set up when the lookaside list was initialized or **ExFreePool**.

On Windows 2000, drivers must use the **-D_WIN2K_COMPAT_SLIST_USAGE** switch to successfully link code that uses **ExFreeToPagedLookasideList**.

For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

ExAllocateFromPagedLookasideList

ExDeletePagedLookasideList

ExInitializePagedLookasideList

PAGED_LOOKASIDE_LIST

ExGetExclusiveWaiterCount function

1/25/2019 • 2 minutes to read • Edit Online

The **ExGetExclusiveWaiterCount** routine returns the number of waiters on exclusive access to a given resource.

Syntax

```
ULONG ExGetExclusiveWaiterCount(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the resource to be tested.

Return Value

ExGetExclusiveWaiterCount returns the number of threads currently waiting to acquire the given resource for exclusive access.

Remarks

ExGetExclusiveWaiterCount can be called to get an estimate of how many other threads might be waiting to modify the data protected by a particular resource variable. The caller cannot assume that the returned value remains constant for any particular interval.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExGetSharedWaiterCount

ExReleaseResourceForThreadLite

ExGetFirmwareEnvironmentVariable function

1/25/2019 • 3 minutes to read • Edit Online

The **ExGetFirmwareEnvironmentVariable** routine gets the value of the specified system firmware environment variable.

Syntax

```
NTSTATUS ExGetFirmwareEnvironmentVariable(
  PUNICODE_STRING VariableName,
  LPGUID          VendorGuid,
  PVOID           Value,
  PULONG          ValueLength,
  PULONG          Attributes
);
```

Parameters

`VariableName`

A pointer to a UNICODE_STRING structure that contains the name of the specified environment variable.

`VendorGuid`

A pointer to a GUID that identifies the vendor associated with the specified environment variable. Environment variables are grouped into namespaces based on their vendor GUIDs. Some hardware platforms might not support vendor GUIDs. On these platforms, all variables are grouped into one, common namespace, and the *VendorGuid* parameter is ignored.

`Value`

A pointer to a caller-allocated buffer to which the routine writes the value of the specified environment variable.

`ValueLength`

A pointer to a location that contains the buffer size. On entry, the location pointed to by this parameter contains the size, in bytes, of the caller-supplied *Value* buffer. Before exiting, the routine writes to this location the size, in bytes, of the variable value. If the routine returns STATUS_SUCCESS, the **ValueLength* output value is the number of bytes of data written to the *Value* buffer. If the routine returns STATUS_BUFFER_TOO_SMALL, **ValueLength* is the required buffer size.

`Attributes`

A pointer to a location to which the routine writes the attributes of the specified environment variable. This parameter is optional and can be set to NULL if the caller does not need the attributes. For more information, see Remarks.

Return Value

ExGetFirmwareEnvironmentVariable returns STATUS_SUCCESS if it is successful. Possible return values include the following error status codes.

RETURN CODE	DESCRIPTION

STATUS_INSUFFICIENT_RESOURCES	Available system resources are insufficient to complete the requested operation.
STATUS_BUFFER_TOO_SMALL	The *Value* buffer is too small.
STATUS_VARIABLE_NOT_FOUND	The requested variable does not exist.
STATUS_INVALID_PARAMETER	One of the parameters is not valid.
STATUS_NOT_IMPLEMENTED	This routine is not supported on this platform.

STATUS_UNSUCCESSFUL	The firmware returned an unrecognized error.

Remarks

System firmware environment variables contain data values that are passed between the boot firmware environment implemented in the hardware platform and the operating-system loaders and other software that runs in the firmware environment.

The set of firmware environment variables that is available in a hardware platform depends on the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based platform, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the Unified Extensible Firmware Interface Specification at the UEFI website. For more information about UEFI and Windows, see UEFI and Windows.

Firmware environment variables are not supported on a legacy BIOS-based platform. Calls to **ExGetFirmwareEnvironmentVariable** always fail on a legacy BIOS-based platform; they also fail if Windows was installed using the legacy BIOS on a platform that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy *VariableName* string (for example, an empty string) and a dummy GUID such as

```
"{00000000-0000-0000-0000-000000000000}"
```

for the *VendorGuid* parameter. On a legacy BIOS-based platform, or on a platform that supports both legacy BIOS and UEFI but in which Windows was installed using the legacy BIOS, the function will fail with STATUS_NOT_IMPLEMENTED. On a UEFI-based platform, the function will fail with an error specific to the firmware, such as STATUS_VARIABLE_NOT_FOUND, to indicate that the dummy GUID namespace does not exist.

If the caller specifies a non-NULL *Attributes* parameter, the routine writes the attributes of the specified system firmware environment variable to the location pointed to by *Attributes*. Version 2.3.1 of the UEFI specification defines the following attributes for firmware environment variables.

VARIABLE NAME	VALUE
EFI_VARIABLE_NON_VOLATILE	0x00000001
EFI_VARIABLE_BOOTSERVICE_ACCESS	0x00000002
EFI_VARIABLE_RUNTIME_ACCESS	0x00000004
EFI_VARIABLE_HARDWARE_ERROR_RECORD	0x00000008
EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS	0x00000010
EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS	0x00000020
EFI_VARIABLE_APPEND_WRITE	0x00000040

These attribute values are defined as flag bits. The value written to the ULONG variable pointed to by *Attributes* is

either zero or the bitwise OR of one or more attributes in the preceding table. For more information, see the UEFI specification at the UEFI website.

If you create a backup datastore, you can use this function to save all the boot settings for the platform so they can be restored by calling the ExSetFirmwareEnvironmentVariable routine if needed.

ExGetFirmwareEnvironmentVariable is the kernel-mode equivalent of the Win32 GetFirmwareEnvironmentVariable function.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ExSetFirmwareEnvironmentVariable

GetFirmwareEnvironmentVariable

UNICODE_STRING

ExGetFirmwareType function

1/25/2019 • 2 minutes to read • Edit Online

Returns the system firmware type.

Syntax

```
FIRMWARE_TYPE ExGetFirmwareType(

);
```

Parameters

This function has no parameters.

Return Value

Returns a FIRMWARE_TYPE-type value that indicates the type of firmware.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	"<=APC_LEVEL"

ExGetPreviousMode function

1/25/2019 • 2 minutes to read • Edit Online

The **ExGetPreviousMode** routine returns the previous processor mode for the current thread.

Syntax

```
KPROCESSOR_MODE ExGetPreviousMode(

);
```

Parameters

This function has no parameters.

Return Value

ExGetPreviousMode returns a **KPROCESSOR_MODE** value, one of **KernelMode** or **UserMode**. This value specifies the previous processor mode for the current thread.

Remarks

If an I/O request can originate either in user mode or kernel mode and the caller passes pointers to data structures used for I/O, the driver must be able to determine whether the caller's pointers are valid in user mode or kernel mode.

If drivers are processing I/O requests using the normal IRP-based I/O dispatch method, they can determine the previous processor mode by checking the *RequestorMode* parameter in the IRP header. (The *RequestorMode* parameter is set by the I/O manager.)

Alternatively, **ExGetPreviousMode** can be used to determine the previous processor mode. This routine is particularly useful in situations where a previous mode parameter is not available, for example, in a file driver that uses fast I/O.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

KeGetCurrentThread

ExGetSharedWaiterCount function

1/25/2019 • 2 minutes to read • Edit Online

The **ExGetSharedWaiterCount** routine returns the number of waiters on shared access to a given resource.

Syntax

```
ULONG ExGetSharedWaiterCount(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the resource to be tested.

Return Value

ExGetSharedWaiterCount returns the number of threads currently waiting to acquire the given resource for shared access.

Remarks

ExGetSharedWaiterCount can be called to get an estimate of how many other threads might be waiting to read the data protected by a particular resource variable. The caller cannot assume that the returned value remains constant for any particular interval.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExGetExclusiveWaiterCount

ExReleaseResourceForThreadLite

ExInitializeDeleteTimerParameters function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInitializeDeleteTimerParameters** routine initializes an EXT_DELETE_PARAMETERS structure.

Syntax

```
void ExInitializeDeleteTimerParameters(
  PEXT_DELETE_PARAMETERS Parameters
);
```

Parameters

`Parameters`

A pointer to the **EXT_DELETE_PARAMETERS** structure that is to be initialized.

Return Value

None.

Remarks

Your driver must call **ExInitializeDeleteTimerParameters** to initialize an **EXT_DELETE_PARAMETERS** structure before the driver passes this structure to the ExDeleteTimer routine. For more information about the member values that **ExInitializeDeleteTimerParameters** writes to the members of this structure, see EXT_DELETE_PARAMETERS.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level.

See Also

EXT_DELETE_PARAMETERS

ExDeleteTimer

ExInitializeFastMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInitializeFastMutex** routine initializes a fast mutex variable, used to synchronize mutually exclusive access by a set of threads to a shared resource.

Syntax

```
void ExInitializeFastMutex(
  PFAST_MUTEX FastMutex
);
```

Parameters

`FastMutex`

A pointer to a caller-allocated FAST_MUTEX structure, which represents the fast mutex, in the nonpaged memory pool. The allocation must be 4-byte aligned on 32-bit platforms, and 8-byte aligned on 64-bit platforms.

Return Value

None

Remarks

ExInitializeFastMutex must be called before any calls to other **Ex*Xxx*FastMutex** routines occur.

Although the caller supplies the storage for the given fast mutex, the **FAST_MUTEX** structure is opaque: that is, its members are reserved for system use.

For better performance, use the **Ex*Xxx*FastMutex** routines instead of the **Ke*Xxx*Mutex** routines. However, a fast mutex cannot be acquired recursively, as a kernel mutex can.

For more information about fast mutexes, see Fast Mutexes and Guarded Mutexes.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExAcquireFastMutex

ExAcquireFastMutexUnsafe

ExReleaseFastMutex

ExReleaseFastMutexUnsafe

ExTryToAcquireFastMutex

FAST_MUTEX

KeInitializeMutex

ExInitializeLookasideListEx function

1/25/2019 • 7 minutes to read • Edit Online

The **ExInitializeLookasideListEx** routine initializes a lookaside list.

Syntax

```
NTSTATUS ExInitializeLookasideListEx(
  PLOOKASIDE_LIST_EX    Lookaside,
  PALLOCATE_FUNCTION_EX Allocate,
  PFREE_FUNCTION_EX     Free,
  POOL_TYPE             PoolType,
  ULONG                 Flags,
  SIZE_T                Size,
  ULONG                 Tag,
  USHORT                Depth
);
```

Parameters

`Lookaside`

A pointer to the LOOKASIDE_LIST_EX structure to initialize. On return, this structure describes an empty lookaside list. The caller must use nonpaged system space for this structure, regardless of whether the entries in the lookaside list are allocated from paged or nonpaged memory. On 64-bit platforms, this structure must be 16-byte aligned.

`Allocate`

A pointer to a caller-supplied LookasideListAllocateEx routine that allocates a new lookaside-list entry. The ExAllocateFromLookasideListEx routine calls this *LookasideListAllocateEx* routine if the lookaside list is empty (contains no entries). This parameter is optional and can be specified as **NULL** if a custom allocation routine is not required. If this parameter is **NULL**, calls to **ExAllocateFromPagedLookasideList** automatically allocate the paged or nonpaged storage (as determined by the *PoolType* parameter) for the new entries.

`Free`

A pointer to a caller-supplied LookasideListFreeEx routine that frees a previously allocated lookaside-list entry. The ExFreeToPagedLookasideList routine calls this *LookasideListFreeEx* routine if the lookaside list is full (that is, the list already contains the maximum number of entries, as determined by the operating system). This parameter is optional and can be specified as **NULL** if a custom deallocation routine is not required. If this parameter is **NULL**, calls to **ExFreeToPagedLookasideList** automatically free the storage for the specified entries.

`PoolType`

Specifies the pool type of the entries in the lookaside list. Set this parameter to a valid POOL_TYPE enumeration value.

`Flags`

Specifies an optional flag value to modify the default behavior of the *LookasideListAllocateEx* routine. Set this parameter to zero or to one of the following EX_LOOKASIDE_LIST_EX_FLAGS_*XXX* flag bits.

FLAG BIT	MEANING
EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL	If the allocation fails, raise an exception.
EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE	If the allocation fails, return **NULL** instead of raising an exception. This flag is intended for use with an allocation routine, such as ExAllocatePoolWithQuotaTag, that charges quotas for pool usage.

These two flag bits are mutually exclusive.

If *Allocate* is **NULL**, set *Flags* to either zero or EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL, but not to EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE. Otherwise, the behavior of the default allocation routine is undefined.

If *Flags* = EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL, the *PoolType* parameter value is bitwise ORed with the POOL_RAISE_IF_ALLOCATION_FAILURE flag bit to form the *PoolType* parameter value that is passed to the *LookasideListAllocateEx* routine. The *LookasideListAllocateEx* routine can pass this *PoolType* value, without modification, to the **ExAllocatePoolWithTag** routine. For more information about the POOL_RAISE_IF_ALLOCATION_FAILURE flag, see ExAllocatePoolWithTag.

If *Flags* = EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE, the *PoolType* parameter value is bitwise ORed with the POOL_QUOTA_FAIL_INSTEAD_OF_RAISE flag bit to form the *PoolType* parameter value that is passed to the *LookasideListAllocateEx* routine. The *LookasideListAllocateEx* routine can pass this *PoolType* value, without modification, to the **ExAllocatePoolWithQuotaTag** routine. For more information about the POOL_QUOTA_FAIL_INSTEAD_OF_RAISE flag, see ExAllocatePoolWithQuotaTag.

`Size`

Specifies the size, in bytes, of each entry in the lookaside list.

`Tag`

Specifies the four-byte pool tag to use to mark the allocated storage for lookaside-list entries. For more information about pool tags, see the description of the *Tag* parameter in **ExAllocatePoolWithTag**.

`Depth`

Reserved. Always set this parameter to zero.

Return Value

ExInitializeLookasideListEx returns STATUS_SUCCESS if the call is successful. Possible return values include the following error code:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER_4	The *PoolType* parameter value is not valid.

STA TUS _IN VAL ID_ PAR AM ETE R_5	The *Flags* parameter value is not valid.

Remarks

A driver must call this routine to initialize a lookaside list before the driver can begin to use the list. A lookaside list is a pool of fixed-size buffers that the driver can manage locally to reduce the number of calls to system allocation routines and, thereby, to improve performance. The buffers are stored as entries in the lookaside list. All entries in the list are of the same, uniform size, which is specified by the *Size* parameter.

After **ExInitializeLookasideListEx** returns, the lookaside list is initialized but contains no entries. When a client calls the ExAllocateFromLookasideListEx routine to request an entry, this routine determines that the lookaside list is empty and calls the driver-supplied LookasideListAllocateEx routine to dynamically allocate storage for a new entry. Additional entries might be allocated in response to similar requests from clients. Later, when clients call the ExFreeToLookasideListEx to release these entries, this routine inserts the entries into the lookaside list. If the number of entries in the list reaches a limit that is determined by the operating system, **ExFreeToLookasideListEx** ceases to add further entries to the list and instead passes these entries to the driver-supplied LookasideListFreeEx routine to be freed.

If the driver does not supply *LookasideListAllocateEx* and *LookasideListFreeEx* routines, the **ExAllocateFromLookasideListEx** and **ExFreeToLookasideListEx** routines use default allocation and deallocation routines instead.

There is no benefit to providing *LookasideListAllocateEx* and *LookasideListFreeEx* routines that do nothing but call ExAllocatePoolWithTag and ExFreePool. The same effect can be achieved with better performance by simply setting the *Allocate* and *Free* parameters to **NULL**.

Before a driver unloads, it must explicitly free any lookaside lists it created. Failure to do so is a serious programming error. Call the ExDeleteLookasideListEx routine to free a lookaside list. This routine frees the storage for any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

The operating system keeps track of all lookaside lists that are currently in use. As both the amount of available nonpaged memory and the demand for lookaside list entries vary over time, the operating system dynamically adjusts its limits for the maximum number of entries in each nonpaged lookaside list.

In Windows 2000 and later versions of Windows, a lookaside list that contains paged or nonpaged entries can be described by a PAGED_LOOKASIDE_LIST or NPAGED_LOOKASIDE_LIST structure, respectively.

For more information about lookaside lists, see Using Lookaside Lists.

Callers of **ExInitializeLookasideListEx** can be running at IRQL <= DISPATCH_LEVEL, but are typically running at IRQL = PASSIVE_LEVEL.

Examples

The driver-supplied *LookasideListAllocateEx* and *LookasideListFreeEx* routines both receive *Lookaside* parameters that point to the **LOOKASIDE_LIST_EX** structure that describes the lookaside list. The routines can use this parameter to access private data that the driver has associated with the lookaside list. For example, the driver might allocate an instance of the following structure to collect private data for each lookaside list that it creates:

```
typedef struct
{
  ULONG NumberOfAllocations;  // number of entries allocated
  ULONG NumberOfFrees;        // number of entries freed
  LOOKASIDE_LIST_EX LookasideField;
} MY_PRIVATE_DATA;
```

The driver can initialize a lookaside list as shown in the following code example:

```
#define ENTRY_SIZE  256
#define MY_POOL_TAG  'tsLL'
MY_PRIVATE_DATA *MyContext;
NTSTATUS status = STATUS_SUCCESS;

MyContext = ExAllocatePoolWithTag(NonPagedPool, sizeof(MY_PRIVATE_DATA), MY_POOL_TAG);
if (MyContext)
{
    MyContext.NumberOfAllocations = 0;
    MyContext.NumberOfFrees = 0;

    status = ExInitializeLookasideListEx(
                &MyContext.LookasideField,
                MyLookasideListAllocateEx,
                MyLookasideListFreeEx,
                NonPagedPool,
                0,
                ENTRY_SIZE,
                MY_POOL_TAG,
                0);
}
else
{
    status = STATUS_INSUFFICIENT_RESOURCES;
}
```

The following code example shows how the *LookasideListAllocateEx* routine can use its *Lookaside* parameter to access the private data that is associated with the lookaside list:

```
PVOID
  MyLookasideListAllocateEx(
    __in POOL_TYPE  PoolType,
    __in SIZE_T  NumberOfBytes,
    __in ULONG  Tag,
    __inout PLOOKASIDE_LIST_EX  Lookaside)
{
    MY_PRIVATE_DATA *MyContext;
    PVOID NewEntry;

    MyContext = CONTAINING_RECORD(Lookaside, MY_PRIVATE_DATA, LookasideField);

    NewEntry = ExAllocatePoolWithTag(PoolType, NumberOfBytes, Tag);

    if (NewEntry)
    {
        MyContext->NumberOfAllocations = InterlockedIncrement(MyContext->NumberOfAllocations);
    }

    return NewEntry;
}
```

The CONTAINING_RECORD macro is defined in the Ntdef.h header file. The *LookAsideListFreeEx* routine can similarly use its *Lookaside* parameter to access private data. After the `MyLookasideListAllocateEx` routine in this example returns, **ExAllocateFromLookasideListEx** inserts the buffer pointed to by the `NewEntry` variable into the lookaside list. To make this insertion operation thread-safe, **ExAllocateFromLookasideListEx** synchronizes its access of the lookaside list with other list insertion and removal operations that might be performed by other threads. Similarly, when **ExFreeFromLookasideListEx** removes a buffer from the lookaside list, it synchronizes its access to the list. **ExAllocateFromLookasideListEx** and **ExFreeFromLookasideListEx** do not synchronize their calls to driver-supplied LookasideListAllocateEx and LookasideListFreeEx routines. Thus, if the `MyLookasideListAllocateEx` and `MyLookasideListFreeEx` routines in the preceding code examples must be thread-safe, the driver must provide the necessary synchronization. The example routine, `MyLookasideListAllocateEx`, synchronizes its access of the `MyContext->NumberOfAllocations` variable with other threads that might increment and decrement this variable. To provide this synchronization, `MyLookasideListAllocateEx` calls the InterlockedIncrement routine to atomically increment this variable. Similarly, the `MyLookasideListFreeEx` routine (not shown) can call the InterlockedDecrement routine to atomically decrement this variable. However, if the sole purpose of the `MyContext->NumberOfAllocations` variable in the preceding code example is simply to gather statistics on lookaside list allocations, atomic increments and decrements are hardly necessary. In this case, the remote possibility of a missed increment or decrement should not be a concern. For more information about thread safety for lookaside lists, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

ExAllocateFromLookasideListEx

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExDeleteLookasideListEx

ExFreePool

ExFreeToLookasideListEx

InterlockedDecrement

InterlockedIncrement

LOOKASIDE_LIST_EX

LookasideListAllocateEx

LookasideListFreeEx

NPAGED_LOOKASIDE_LIST

PAGED_LOOKASIDE_LIST

POOL_TYPE

ExInitializeNPagedLookasideList function

1/25/2019 • 4 minutes to read • Edit Online

The **ExInitializeNPagedLookasideList** routine initializes a lookaside list for nonpaged entries of the specified size.

Syntax

```
void ExInitializeNPagedLookasideList(
  PNPAGED_LOOKASIDE_LIST Lookaside,
  PALLOCATE_FUNCTION     Allocate,
  PFREE_FUNCTION         Free,
  ULONG                  Flags,
  SIZE_T                 Size,
  ULONG                  Tag,
  USHORT                 Depth
);
```

Parameters

`Lookaside`

A pointer to the NPAGED_LOOKASIDE_LIST structure to initialize. The caller must use nonpaged system space for the structure. On 64-bit platforms, this structure must be 16-byte aligned.

`Allocate`

A pointer to either a caller-supplied function for allocating an entry when the lookaside list is empty, or to **NULL**. If non-**NULL**, the pointer is to a function with the prototype:

```
PVOID XxxAllocate(
  __in POOL_TYPE  PoolType,           // NonPagedPool
  __in SIZE_T  NumberOfBytes,         // value of Size
  __in ULONG  Tag                     // value of Tag
);
```

If the *Allocate* parameter is **NULL**, subsequent calls to ExAllocateFromNPagedLookasideList automatically allocate entries whenever the lookaside list is empty.

`Free`

A pointer to either a caller-supplied function for freeing an entry whenever the lookaside list is full, or to **NULL**. If non-**NULL**, the pointer is to a function with the prototype:

```
VOID XxxFree(
  __in PVOID  Buffer
);
```

If the *Free* parameter is **NULL**, subsequent calls to ExFreeToNPagedLookasideList automatically release the given entry back to nonpaged pool whenever the list is full, that is, currently holding the system-determined maximum number of entries.

`Flags`

Starting in Windows 8, this parameter specifies an optional flag value to modify the default behavior of the **ExInitializeNPagedLookasideList** routine. Compatible flag bits include the following.

FLAG BIT	MEANING
POOL_NX_ALLOCATION	Allocate non-executable memory.
POOL_RAISE_IF_ALLOCATION_FAILURE	If the allocation fails, raise an exception.

Before Windows 8, this parameter is not used and must be zero.

`Size`

Specifies the size, in bytes, for each nonpaged entry to be allocated subsequently. This parameter must not be less than the required minimum size, LOOKASIDE_MINIMUM_BLOCK_SIZE, which is defined in the Wdm.h header file.

`Tag`

Specifies the pool tag to use when allocating lookaside list entries. For more information about pool tags, see the *Tag* parameter of ExAllocatePoolWithTag.

`Depth`

Reserved. Must be zero.

Return Value

None

Remarks

After calling **ExInitializeNPagedLookasideList**, memory blocks of the caller-specified *Size* can be allocated from and freed to the lookaside list with calls to ExAllocateFromNPagedLookasideList and ExFreeToNPagedLookasideList, respectively. Such dynamically allocated and freed entries can be any data structure or fixed-size buffer that the caller uses while the system is running, particularly if the caller cannot predetermine how many such entries will be in use at any given moment. The layout and contents of each fixed-size entry are caller-determined.

ExInitializeNPagedLookasideList initializes the system state to track usage of the given lookaside list, as follows:

- Zero-initializes the counters to be maintained for entries.
- Stores the entry points of the caller-supplied ***Xxx*Allocate** and ***Xxx*Free** routines, if any, or sets these entry points to ExAllocatePoolWithTag and ExFreePool, respectively.

- Initializes a system spin lock to control allocations from and frees to the lookaside list in a multiprocessor-safe manner if necessary.
- Stores the caller-supplied entry *Size* and list *Tag*.
- Sets up the system-determined limits (minimum and maximum) on the number of entries to be held in the lookaside list, which can be adjusted subsequently if system-wide demand for entries is higher or lower than anticipated.
- Sets up the system-determined flags, which control the type of memory from which entries will be allocated subsequently.

The system maintains a set of all lookaside lists currently in use. As demand for lookaside list entries and on available nonpaged memory varies while the system runs, the system adjusts its limits for the number of entries to be held in each nonpaged lookaside list dynamically.

Drivers must always explicitly free any lookaside lists they create before unloading. To do otherwise is a serious programming error. Use ExDeleteNPagedLookasideList to free the list.

ExInitializeNPagedLookasideList sets up the opaque list head at the caller-supplied location but preallocates no memory for list entries. Subsequently, the initial entries are allocated dynamically as calls to **ExAllocateFromNPagedLookasideList** occur, and these initial entries are held in the lookaside list as reciprocal calls to **ExFreeToNPagedLookasideList** occur. Entries collect in the given lookaside list until the system-determined maximum is reached, whereupon any additional entries are returned to nonpaged pool as they are freed. If the list becomes empty, allocate requests are satisfied by the ***Xxx*Allocate** function specified at list initialization or by ExAllocatePoolWithTag.

It is more efficient to pass **NULL** pointers for the *Allocate* and *Free* parameters of **ExInitializeNPagedLookasideList** whenever the user of a lookaside list does nothing more than allocate and release fixed-size entries. However, any component that uses a lookaside list might supply these functions to do additional caller-determined processing, such as tracking its own dynamic memory usage by maintaining state about the number of entries it allocates and frees.

If the caller of **ExInitializeNPagedLookasideList** supplies an ***Xxx*Allocate** function, that routine must allocate entries for the lookaside list using the given input parameters when it calls **ExAllocatePoolWithTag.**

Starting with Windows Vista, a similar routine, ExInitializeLookasideListEx, initializes a lookaside list that is described by a LOOKASIDE_LIST_EX structure. Unlike the ***Xxx*Allocate** and ***Xxx*Free** routines for a lookaside list that uses an **NPAGED_LOOKASIDE_LIST** structure, the allocation and deallocation routines for a lookaside list that uses the **LOOKASIDE_LIST_EX** structure receive a context pointer as an input parameter. These routines can use this context to store private data for the lookaside list. If your driver is intended to run only in Windows Vista and later versions of Windows, consider using **ExInitializeLookasideListEx** instead of **ExInitializeNPagedLookasideList**. For more information, see Using Lookaside Lists.

Callers of **ExInitializeNPagedLookasideList** can be running at IRQL <= DISPATCH_LEVEL, but are typically running at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

ExAllocateFromNPagedLookasideList

ExAllocatePoolWithTag

ExDeleteNPagedLookasideList

ExFreePool

ExFreeToNPagedLookasideList

ExInitializeLookasideListEx

ExInitializePagedLookasideList

LOOKASIDE_LIST_EX

NPAGED_LOOKASIDE_LIST

ExInitializePagedLookasideList function

1/25/2019 • 4 minutes to read • Edit Online

The **ExInitializePagedLookasideList** routine initializes a lookaside list for pageable entries of the specified size.

Syntax

```
void ExInitializePagedLookasideList(
  PPAGED_LOOKASIDE_LIST Lookaside,
  PALLOCATE_FUNCTION    Allocate,
  PFREE_FUNCTION        Free,
  ULONG                 Flags,
  SIZE_T                Size,
  ULONG                 Tag,
  USHORT                Depth
);
```

Parameters

`Lookaside`

A pointer to the PAGED_LOOKASIDE_LIST structure to initialize. The caller must use nonpaged system space for the structure, even though the entries in this lookaside list will be allocated from pageable memory. On 64-bit platforms, this structure must be 16-byte aligned.

`Allocate`

A pointer to either a caller-supplied function for allocating an entry when the lookaside list is empty, or to **NULL**. If non-**NULL**, the pointer is to a function with the prototype:

```
PVOID XxxAllocate(
  __in POOL_TYPE  PoolType,           // PagedPool
  __in SIZE_T  NumberOfBytes,         // value of Size
  __in ULONG  Tag                     // value of Tag
);
```

If the *Allocate* parameter is **NULL**, subsequent calls to **ExAllocateFromPagedLookasideList** automatically allocate entries whenever the lookaside list is empty.

`Free`

A pointer to either a caller-supplied function for freeing an entry whenever the lookaside list is full, or to **NULL**. If non-**NULL**, the pointer is to a function with the prototype:

```
VOID XxxFree(
  __in PVOID  Buffer
);
```

If the *Free* parameter is **NULL**, subsequent calls to **ExFreeToPagedLookasideList** automatically release the given entry back to paged pool whenever the list is full, that is, currently holding the system-determined maximum number of entries.

`Flags`

Starting in Windows 8, this parameter specifies an optional flag value to modify the default behavior of the **ExInitializePagedLookasideList** routine. Compatible flag bits include the following.

FLAG BIT	MEANING
POOL_NX_ALLOCATION	Allocate non-executable memory.
POOL_RAISE_IF_ALLOCATION_FAILURE	If the allocation fails, raise an exception.

Before Windows 8, this parameter is not used and must be zero.

`Size`

Specifies the size in bytes of each entry in the lookaside list.

`Tag`

Specifies the pool tag to use when allocating lookaside list entries. For more information about pool tags, see the *Tag* parameter of ExAllocatePoolWithTag.

`Depth`

Reserved. Must be zero.

Return Value

None

Remarks

After calling **ExInitializePagedLookasideList**, blocks of the caller-specified *Size* can be allocated from and freed to the lookaside list with calls to ExAllocateFromPagedLookasideList and ExFreeToPagedLookasideList, respectively. Such dynamically allocated and freed entries can be any data structure or fixed-size buffer that the caller uses while the system is running, particularly if the caller cannot predetermine how many such entries will be in use at any given moment. The layout and contents of each fixed-size entry are caller-determined.

ExInitializePagedLookasideList initializes the system state to track usage of the given lookaside list, as follows:

- Zero-initializes the counters to be maintained for entries.
- Stores the entry points of the caller-supplied ***Xxx*Allocate** and ***Xxx*Free** routines, if any, or sets these entry points to ExAllocatePoolWithTag and ExFreePool, respectively.
- Initializes a system spin lock to control allocations from and frees to the lookaside list in a multiprocessor-safe manner if necessary.

- Stores the caller-supplied entry *Size* and list *Tag*.
- Sets up the system-determined limits (minimum and maximum) on the number of entries to be held in the lookaside list, which can be adjusted subsequently if system-wide demand for entries is higher or lower than anticipated.
- Sets up the system-determined flags, which control the type of memory from which entries will be allocated subsequently.

The system maintains a set of all lookaside lists in use. As demand for lookaside list entries and on available paged memory varies while the system runs, the system adjusts its limits for the number of entries to be held in each paged lookaside list dynamically.

Drivers must always use explicitly free any lookaside lists they create before unloading. To do otherwise is a serious programming error. Use ExDeletePagedLookasideList to free the list.

ExInitializePagedLookasideList sets up the opaque list head at the caller-supplied location but preallocates no memory for list entries. Subsequently, the initial entries are allocated dynamically as calls to **ExAllocateFromPagedLookasideList** occur, and these initial entries are held in the lookaside list as reciprocal calls to **ExFreeToPagedLookasideList** occur. Entries collect in the given lookaside list until the system-determined maximum is reached, whereupon any additional entries are returned to paged pool as they are freed. If the list becomes empty, allocate requests are satisfied by the ***Xxx*Allocate** function specified at list initialization or by **ExAllocatePoolWithTag**.

It is more efficient to pass **NULL** pointers for the *Allocate* and *Free* parameters of **ExInitializePagedLookasideList** whenever the user of a lookaside list does nothing more than allocate and release fixed-size entries. However, any component that uses a lookaside list might supply these functions to do additional caller-determined processing, such as tracking its own dynamic memory usage by maintaining state about the number of entries it allocates and frees.

If the caller of **ExInitializePagedLookasideList** supplies an ***Xxx*Allocate** function, that function must allocate entries for the lookaside list using the given input parameters when it calls **ExAllocatePoolWithTag.**

Starting with Windows Vista, a similar routine, ExInitializeLookasideListEx, initializes a lookaside list that is described by a LOOKASIDE_LIST_EX structure. Unlike the ***Xxx*Allocate** and ***Xxx*Free** routines for a lookaside list that uses a **PAGED_LOOKASIDE_LIST** structure, the allocation and deallocation routines for a lookaside list that uses the **LOOKASIDE_LIST_EX** structure receive a context pointer as an input parameter. These routines can use this context to store private data for the lookaside list. If your driver is intended to run only in Windows Vista and later versions of Windows, consider using **ExInitializeLookasideListEx** instead of **ExInitializePagedLookasideList**. For more information, see Using Lookaside Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ExAllocateFromPagedLookasideList

ExAllocatePoolWithTag

ExDeletePagedLookasideList

ExFreePool

ExFreeToPagedLookasideList

ExInitializeLookasideListEx

ExInitializeNPagedLookasideList

LOOKASIDE_LIST_EX

PAGED_LOOKASIDE_LIST

ExInitializeResourceLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInitializeResourceLite** routine initializes a resource variable.

Syntax

```
NTSTATUS ExInitializeResourceLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the caller-supplied storage, which must be at least **sizeof**(**ERESOURCE**), for the resource variable being initialized. The storage must be 4-byte aligned on 32-bit platforms, and 8-byte aligned on 64-bit platforms.

Return Value

ExInitializeResourceLite returns STATUS_SUCCESS.

Remarks

The storage for **ERESOURCE** must be allocated from nonpaged pool.

The resource variable can be used for synchronization by a set of threads. Although the caller provides the storage for the resource variable, the **ERESOURCE** structure is opaque: that is, its members are reserved for system use.

Call **ExDeleteResourceLite** before freeing the memory for the resource.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExConvertExclusiveToSharedLite

ExDeleteResourceLite

ExIsResourceAcquiredExclusiveLite

ExIsResourceAcquiredSharedLite

ExReinitializeResourceLite

ExReleaseResourceForThreadLite

ExInitializeRundownProtection function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInitializeRundownProtection** routine initializes run-down protection on a shared object.

Syntax

```
void ExInitializeRundownProtection(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to a caller-allocated EX_RUNDOWN_REF structure that is to be initialized. The allocation must be 4-byte aligned on 32-bit platforms, and 8-byte aligned on 64-bit platforms. For more information, see Remarks.

Return Value

None.

Remarks

This routine is called by the driver that owns an object that resides in shared memory and is that accessed by other drivers.

ExInitializeRundownProtection must be called to initialize an **EX_RUNDOWN_REF** structure before it is passed as a parameter to any other run-down protection routine.

The run-down protection routines use the **EX_RUNDOWN_REF** structure to track the run-down status of a shared object that is associated with the structure. This status information includes a count of instances of run-down protection that are currently in effect on the object. **ExInitializeRundownProtection** initializes this count to zero.

After an **EX_RUNDOWN_REF** structure is initialized, other drivers that access the associated object can call the ExAcquireRundownProtection and ExReleaseRundownProtection routines to acquire and release run-down protection on the object.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	wdm.h (include Wdm.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExReleaseRundownProtection

ExInitializeSetTimerParameters function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInitializeSetTimerParameters** routine initializes an EXT_SET_PARAMETERS structure.

Syntax

```
void ExInitializeSetTimerParameters(
  PEXT_SET_PARAMETERS Parameters
);
```

Parameters

`Parameters`

A pointer to the **EXT_SET_PARAMETER** structure that is to be initialized.

Return Value

None.

Remarks

Your driver must call **ExInitializeSetTimerParameters** to initialize an **EXT_SET_PARAMETERS** structure before the driver passes this structure to the ExSetTimer routine. For more information about the member values that **ExInitializeSetTimerParameters** writes to the members of this structure, see EXT_SET_PARAMETERS.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level.

See Also

EXT_SET_PARAMETERS

ExSetTimer

ExInterlockedAddLargeInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedAddLargeInteger** routine adds a large integer value to the specified variable as an atomic operation.

Syntax

```
LARGE_INTEGER ExInterlockedAddLargeInteger(
  PLARGE_INTEGER Addend,
  LARGE_INTEGER  Increment,
  PKSPIN_LOCK    Lock
);
```

Parameters

`Addend`

A pointer to the variable to be adjusted by the *Increment* value.

`Increment`

Specifies a value to be added to *Addend*.

`Lock`

A pointer to a spin lock to be used to synchronize access to *Addend*.

Return Value

ExInterlockedAddLargeInteger returns the initial value of the *Addend*parameter.

Remarks

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity on symmetric multiprocessor machines. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked*Xxx***.

The *Lock* passed to **ExInterlockedAddLargeInteger** is used to assure that the add operation on *Addend* is atomic with respect to any other operations on the same value which synchronize with this same spin lock.

ExInterlockedAddLargeInteger masks interrupts. Consequently, it can be used for synchronization between an ISR and other device driver code, provided that the same *Lock* is never reused in a call to a routine that runs at IRQL = DISPATCH_LEVEL.

Note that calls to **Interlocked*Xxx*** are guaranteed to be atomic with respect to other **Interlocked*Xxx*** calls without caller-supplied spin locks.

Callers of **ExInterlockedAddLargeInteger** run at any IRQL. The storage for the *Addend* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Not available in Windows 98 or Windows Me.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInterlockedAddUlong

InterlockedDecrement

InterlockedIncrement

KeInitializeSpinLock

ExInterlockedAddLargeStatistic macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedAddLargeStatistic** routine performs an interlocked addition of a ULONG increment value to a LARGE_INTEGER variable.

Syntax

```
void ExInterlockedAddLargeStatistic(
   Addend,
   Increment
);
```

Parameters

`Addend`

A pointer to the LARGE_INTEGER variable that is incremented by the value of *Increment*.

`Increment`

Specifies a ULONG value that is added to the variable that *Addend* points to.

Return Value

None

Remarks

Support routines that do interlocked operations must not cause a page fault. Neither their code nor any of the data they access can cause a page fault without bringing down the system.

ExInterlockedAddLargeStatistic masks interrupts, and can be safely used to synchronize an ISR with other driver code.

ExInterlockedAddLargeStatistic runs at any IRQL. The storage for the *Addend* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

ExInterlockedAddUlong function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedAddUlong** routine adds an unsigned long value to a given unsigned integer as an atomic operation.

Syntax

```
ULONG ExInterlockedAddUlong(
  PULONG      Addend,
  ULONG       Increment,
  PKSPIN_LOCK Lock
);
```

Parameters

`Addend`

A pointer to an unsigned long integer whose value is to be adjusted by the *Increment* value.

`Increment`

Specifies an unsigned long integer to be added.

`Lock`

A pointer to a spin lock to be used to synchronize access to the *Addend*.

Return Value

ExInterlockedAddUlong returns the original (unsummed) value of the *Addend*.

Remarks

Consider using **InterlockedExchangeAdd** instead of this routine. **InterlockedExchangeAdd** can be more efficient because it does not use a spin lock and it is inlined by the compiler.

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity on symmetric multiprocessor machines. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked*Xxx***.

The *Lock* passed to **ExInterlockedAddULong** is used to assure that the add operation on *Addend* is atomic with respect to any other operations on the same value which synchronize with this same spin lock.

ExInterlockedAddUlong masks interrupts. Consequently, it can be used for synchronization between an ISR and other driver code, provided that the same *Lock* is never reused in a call to a routine that runs at IRQL = DISPATCH_LEVEL.

Note that calls to **Interlocked*Xxx*** are guaranteed to be atomic with respect to other **Interlocked*Xxx*** calls without caller-supplied spin locks.

Callers of **ExInterlockedAddUlong** run at any IRQL. The storage for the *Addend* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInterlockedAddLargeInteger

InterlockedDecrement

InterlockedIncrement

KeInitializeSpinLock

ExInterlockedCompareExchange64 macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedCompareExchange64** routine compares one integer variable to another and, if they are equal, sets the first variable to a caller-supplied value.

Syntax

```
void ExInterlockedCompareExchange64(
   Destination,
   Exchange,
   Comperand,
   Lock
);
```

Parameters

`Destination`

A pointer to an integer that will be compared and possibly replaced.

`Exchange`

A pointer to an integer that will replace the one at *Destination* if the comparison results in equality.

`Comperand`

A pointer to an integer with which the value at *Destination* will be compared.

`Lock`

A pointer to a caller-allocated spin-lock that is used if the host system does not support an 8-byte atomic compare-and-exchange operation.

Return Value

This macro wraps **InterlockedCompareExchange64** or **ExfInterlockedCompareExchange64** and assumes the return value of the type LONGLONG.

Remarks

ExInterlockedCompareExchange64 tests and, possibly, replaces the value of a given variable. For most underlying microprocessors, this routine is implemented inline by the compiler to execute as an atomic operation. If a spin lock is used, this routine can only be safely used on nonpaged parameters.

If the *Destination* and *Comparand* are unequal, **ExInterlockedCompareExchange64** simply returns the value of *Destination*.

ExInterlockedCompareExchange64 is atomic only with respect to other **(Ex)Interlocked*Xxx*** calls.

Callers of **ExInterlockedCompareExchange64** can be running at any IRQL. The storage for the *Destination*, Comparand, and *Exchange* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

InterlockedCompareExchange

InterlockedExchange

InterlockedExchangeAdd

ExInterlockedFlushSList macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedFlushSList** routine atomically removes all entries from a sequenced singly linked list.

Syntax

```
void ExInterlockedFlushSList(
   Head
);
```

Parameters

`Head`

Return Value

If there were entries on the specified list, **ExInterlockedFlushSList** returns a pointer to the first **SLIST_ENTRY** structure that was entry on the list; otherwise, it returns NULL.

Remarks

ExInterlockedFlushSList does not delete the SLIST_ENTRY structures that made up the list; it only sets the internal pointer of *ListHead* to the beginning of the list to **NULL**. The driver must free the entries explicitly.

The routine returns a pointer to the first **SLIST_ENTRY** structure that was on the list. The driver can use this pointer to iterate through the entries.

For more information about using this routine to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

Callers of **ExInterlockedFlushSList** can be running at any IRQL. The storage for the *ListHead* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInitializeSListHead

ExInterlockedInsertHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedInsertHeadList** routine atomically inserts an entry at the beginning of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY ExInterlockedInsertHeadList(
  PLIST_ENTRY                  ListHead,
  __drv_aliasesMem PLIST_ENTRY ListEntry,
  PKSPIN_LOCK                  Lock
);
```

Parameters

`ListHead`

A pointer to the LIST_ENTRY structure that serves as the list header.

`ListEntry`

A pointer to the LIST_ENTRY structure that represents the entry to be inserted into the list.

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedInsertHeadList returns a pointer to the first entry of the list before the new entry was inserted. If the list was empty, the routine returns **NULL**.

Remarks

ExInterlockedInsertHeadList performs the same operation as InsertHeadList, but atomically. Do not mix atomic and non-atomic calls on the same list.

For more information about using this routine to implement a doubly linked list, see Singly and Doubly Linked Lists.

The **ExInterlockedInsertHeadList** routine can be called at any IRQL. The storage for the *ListHead* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)
DDI compliance rules	IoAllocateFree, IoReuseIrp, MarkingInterlockedQueuedIrps, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

ExInterlockedInsertTailList

ExInterlockedRemoveHeadList

InitializeListHead

InsertHeadList

KeInitializeSpinLock

ExInterlockedInsertTailList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedInsertTailList** routine atomically inserts an entry at the end of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY ExInterlockedInsertTailList(
  PLIST_ENTRY                  ListHead,
  __drv_aliasesMem PLIST_ENTRY ListEntry,
  PKSPIN_LOCK                  Lock
);
```

Parameters

`ListHead`

A pointer to the LIST_ENTRY structure that serves as the list header.

`ListEntry`

A pointer to the LIST_ENTRY structure that represents the entry to be inserted into the list.

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedInsertTailList returns a pointer to the last entry of the list before the new entry was inserted. If the list was empty, the routine returns **NULL**.

Remarks

ExInterlockedInsertTailList performs the same operation as InsertTailList, but atomically. Do not mix atomic and non-atomic calls on the same list.

For more information about using this routine to implement a doubly linked list, see Singly and Doubly Linked Lists.

The **ExInterlockedInsertTailList** routine can be called at any IRQL. The storage for the *ListHead* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)
DDI compliance rules	IoAllocateFree, IoReuseIrp, MarkingInterlockedQueuedIrps, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

ExInterlockedInsertHeadList

InitializeListHead

InsertTailList

KeInitializeSpinLock

ExInterlockedPopEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedPopEntryList** routine atomically removes an entry from the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
PSINGLE_LIST_ENTRY ExInterlockedPopEntryList(
  PSINGLE_LIST_ENTRY ListHead,
  PKSPIN_LOCK        Lock
);
```

Parameters

`ListHead`

A pointer to the SINGLE_LIST_ENTRY structure that serves as the list header.

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedPopEntryList returns a pointer to the SINGLE_LIST_ENTRY structure removed from the list. If the list was empty, the routine returns **NULL**.

Remarks

ExInterlockedPopEntryList performs the same operation as PopEntryList, but atomically. Do not mix atomic and non-atomic calls on the same list.

For more information about using this routine to implement a singly linked list, see Singly and Doubly Linked Lists.

The **ExInterlockedPopEntryList** routine can be called at any IRQL. The storage for the *ListHead* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInitializeSListHead

ExInterlockedPopEntrySList

ExInterlockedPushEntryList

KeInitializeSpinLock

PopEntryList

ExInterlockedPopEntrySList macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedPopEntrySList** routine atomically removes the first entry from a sequenced singly linked list.

Syntax

```
void ExInterlockedPopEntrySList(
   Head,
   Lock
);
```

Parameters

`Head`

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedPopEntrySList returns a pointer to the first **SLIST_ENTRY** structure on the list. If the list was empty, the routine returns NULL.

Remarks

For more information about using this routine to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

On Windows 2000, drivers must use the **-D_WIN2K_COMPAT_SLIST_USAGE** switch to successfully link code that uses **ExInterlockedPopEntrySList**.

ExInterlockedPopEntrySList can be called at any IRQL. The storage for the *ListHead* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInitializeSListHead

ExInterlockedPushEntrySList

ExInterlockedRemoveHeadList

ExQueryDepthSList

KeInitializeSpinLock

ExInterlockedPushEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedPushEntryList** routine atomically inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
PSINGLE_LIST_ENTRY ExInterlockedPushEntryList(
  PSINGLE_LIST_ENTRY                  ListHead,
  __drv_aliasesMem PSINGLE_LIST_ENTRY ListEntry,
  PKSPIN_LOCK                         Lock
);
```

Parameters

`ListHead`

A pointer to the SINGLE_LIST_ENTRY structure that serves as the list header.

`ListEntry`

A pointer to the SINGLE_LIST_ENTRY structure that represents the entry to be inserted into the list.

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedPushEntryList returns a pointer to the first entry of the list before the new entry was inserted. If the list was empty, the routine returns **NULL**.

Remarks

ExInterlockedPushEntryList performs the same operation as PushEntryList, but atomically. Do not mix atomic and non-atomic calls on the same list.

For more information about using this routine to implement a singly linked list, see Singly and Doubly Linked Lists.

The **ExInterlockedPushEntryList** routine can be called at any IRQL. The storage for the *ListHead* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)
DDI compliance rules	IoAllocateFree, IoReuseIrp, MarkingInterlockedQueuedIrps, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

ExInitializeSListHead

ExInterlockedInsertTailList

ExInterlockedPopEntryList

ExInterlockedPushEntrySList

KeInitializeSpinLock

PushEntryList

ExInterlockedPushEntrySList macro

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedPushEntrySList** routine atomically inserts an entry at the beginning of a sequenced singly linked list.

Syntax

```
void ExInterlockedPushEntrySList(
   Head,
   Entry,
   Lock
);
```

Parameters

`Head`

`Entry`

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

None

Remarks

For more information about using this routine to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

On Windows 2000, drivers must use the **-D_WIN2K_COMPAT_SLIST_USAGE** switch to successfully link code that uses **ExInterlockedPushEntrySList**.

The **ExInterlockedPushEntrySList** routine can be called at any IRQL. The storage for the *ListHead* parameter and the list entries must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInitializeSListHead

ExInterlockedInsertTailList

ExInterlockedPopEntrySList

ExQueryDepthSList

KeInitializeSpinLock

ExInterlockedRemoveHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExInterlockedRemoveHeadList** routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY ExInterlockedRemoveHeadList(
  PLIST_ENTRY ListHead,
  PKSPIN_LOCK Lock
);
```

Parameters

`ListHead`

A pointer to the LIST_ENTRY structure that serves as the list header.

`Lock`

A pointer to a **KSPIN_LOCK** structure that serves as the spin lock used to synchronize access to the list. The storage for the spin lock must be resident and must have been initialized by calling KeInitializeSpinLock. You must use this spin lock only with the **ExInterlocked*Xxx*List** routines.

Return Value

ExInterlockedRemoveHeadList returns a pointer to the LIST_ENTRY structure removed from the list. If the list was empty, the routine returns **NULL**.

Remarks

ExInterlockedRemoveHeadList performs a similar operation as RemoveHeadList, but atomically. Do not mix atomic and non-atomic calls on the same list.

For more information about using this routine to implement a doubly linked list, see Singly and Doubly Linked Lists.

The **ExInterlockedRemoveHeadList** routine can be called at any IRQL. The storage for the *ListHead* parameter must be resident at all IRQLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInterlockedInsertHeadList

ExInterlockedInsertTailList

InitializeListHead

KeInitializeSpinLock

RemoveHeadList

ExIsProcessorFeaturePresent function

1/25/2019 • 2 minutes to read • Edit Online

The **ExIsProcessorFeaturePresent** routine queries for the existence of a specified processor feature.

Syntax

```
BOOLEAN ExIsProcessorFeaturePresent(
  ULONG ProcessorFeature
);
```

Parameters

`ProcessorFeature`

Specifies one of the following constant values:

PF_FLOATING_POINT_PRECISION_ERRATA

The processor has the Pentium floating-point divide bug.

PF_FLOATING_POINT_EMULATED

The processor does not have floating-point hardware.

PF_COMPARE_EXCHANGE_DOUBLE

The processor has an 8-byte, memory-locked compare and exchange (CMPXCHG8B) instruction.

PF_MMX_INSTRUCTIONS_AVAILABLE

The processor supports MMX instructions in hardware.

PF_XMMI_INSTRUCTIONS_AVAILABLE

The processor supports SSE instructions in hardware.

PF_3DNOW_INSTRUCTIONS_AVAILABLE

The processor supports AMD 3DNow instructions.

PF_RDTSC_INSTRUCTION_AVAILABLE

The processor supports a read-timestamp-counter (RDTSC) instruction.

PF_PAE_ENABLED

The processor implements Physical Address Extension (PAE) support.

PF_XMMI64_INSTRUCTIONS_AVAILABLE

The processor supports SSE2 instructions in hardware. This parameter value is supported only in Windows XP and later versions of Windows.

PF_SSE_DAZ_MODE_AVAILABLE

The processor supports the denormals-are-zero (DAZ) mode for SSE instructions. This parameter value is supported only in Windows Vista and later versions of Windows.

PF_NX_ENABLED

Data execution prevention is enabled. This parameter value is supported only in Windows Vista and later versions of Windows.

PF_SSE3_INSTRUCTIONS_AVAILABLE

The processor supports SSE3 instructions. This parameter value is supported only in Windows Vista and later

versions of Windows.

PF_COMPARE_EXCHANGE128

The atomic compare and exchange 128-bit operation (CMPXCHG16B) is available. This parameter value is supported only in Windows Vista and later versions of Windows.

PF_COMPARE64_EXCHANGE128

The atomic compare 64-bit and exchange 128-bit operation (CMP8XCHG16) is available. This parameter value is supported only in Windows Vista and later versions of Windows.

PF_XSAVE_ENABLED

The processor supports the XSAVE and XRSTOR instructions. This parameter value is supported only in Windows 7 and later versions of Windows.

Return Value

ExIsProcessorFeaturePresent returns **TRUE** if the specified processor feature is present; otherwise, it returns **FALSE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlExPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

ExIsResourceAcquiredExclusiveLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExIsResourceAcquiredExclusiveLite** routine returns whether the current thread has exclusive access to a given resource.

Syntax

```
BOOLEAN ExIsResourceAcquiredExclusiveLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the resource to be queried.

Return Value

ExIsResourceAcquiredExclusiveLite returns **TRUE** if the caller already has exclusive access to the given resource.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceExclusiveLite

ExIsResourceAcquiredSharedLite

ExIsResourceAcquiredSharedLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExIsResourceAcquiredSharedLite** routine returns whether the current thread has access (either shared or exclusive) to a given resource.

Syntax

```
ULONG ExIsResourceAcquiredSharedLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the resource to be queried.

Return Value

ExIsResourceAcquiredSharedLite returns the number of times the caller has acquired the given resource for shared or exclusive access.

Remarks

The system considers exclusive access to be a subset of shared access. Therefore, a thread that has exclusive access to a resource also has shared access to the resource.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExIsResourceAcquiredExclusiveLite

ExIsSoftBoot function

1/25/2019 • 2 minutes to read • Edit Online

Determines whether the system has gone through a soft restart.

Syntax

```
BOOLEAN ExIsSoftBoot(

);
```

Parameters

This function has no parameters.

Return Value

TRUE indicates a soft restart; FALSE otherwise.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	"<=APC_LEVEL"

ExLocalTimeToSystemTime function

1/25/2019 • 2 minutes to read • Edit Online

The **ExLocalTimeToSystemTime** routine converts a system time value for the current time zone to an unbiased, GreenGMT value.

Syntax

```
void ExLocalTimeToSystemTime(
  PLARGE_INTEGER LocalTime,
  PLARGE_INTEGER SystemTime
);
```

Parameters

`LocalTime`

A pointer to a variable set to the locale-specific time.

`SystemTime`

A pointer to the returned value for GMT system time.

Return Value

None

Remarks

ExLocalTimeToSystemTime adds the time-zone bias at the current locale to compute the corresponding GMT system time value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ExSystemTimeToLocalTime

ExNotifyCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **ExNotifyCallback** routine causes all callback routines registered for the given object to be called.

Syntax

```
void ExNotifyCallback(
  PVOID CallbackObject,
  PVOID Argument1,
  PVOID Argument2
);
```

Parameters

`CallbackObject`

A pointer to the callback object for which all registered callback routines will be called.

`Argument1`

Specifies the parameter that is passed as *Argument1* of the callback routine.

`Argument2`

Specifies the parameter that is passed as *Argument2* of the callback routine.

Return Value

None

Remarks

Driver writers must not call **ExNotifyCallback** for any of the system-defined callback objects listed in **ExCreateCallback**.

The system calls callback routines in order of their registration.

For more information about callback objects, see Callback Objects.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL. The system calls all registered callback routines at the caller's IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

ExCreateCallback

ExRegisterCallback

ExQueryDepthSList function

1/25/2019 • 2 minutes to read • Edit Online

The **ExQueryDepthSList** routine returns the number of entries currently in a given sequenced singly linked list.

Syntax

```
USHORT ExQueryDepthSList(
  PSLIST_HEADER SListHead
);
```

Parameters

`SListHead`

A pointer to the SLIST_HEADER structure that serves as the header for the sequenced singly linked list. *SListHead* must have been initialized by calling ExInitializeSListHead.

Return Value

ExQueryDepthSList returns the current number of entries in the list.

Remarks

For more information about using this routine to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ExInitializeSListHead

ExInterlockedPopEntrySList

ExInterlockedPushEntrySList

ExQueryTimerResolution function

1/25/2019 • 2 minutes to read • Edit Online

The **ExQueryTimerResolution** routine reports the range of timer resolutions that are supported by the system clock.

Syntax

```
void ExQueryTimerResolution(
  PULONG MaximumTime,
  PULONG MinimumTime,
  PULONG CurrentTime
);
```

Parameters

`MaximumTime`

A pointer to a location to which the routine writes the maximum time interval, in 100-nanosecond units, between successive ticks of the system clock. A *tick* is an interrupt caused by the system clock timer.

`MinimumTime`

A pointer to a location to which the routine writes the minimum time interval, in 100-nanosecond units, between successive ticks of the system clock.

`CurrentTime`

A pointer to a location to which the routine writes the current time interval, in 100-nanosecond units, between successive ticks of the system clock.

Return Value

None.

Remarks

If your driver calls the ExSetTimerResolution routine to change the time interval between successive system clock interrupts, the driver can first call **ExQueryTimerResolution** to determine the range of intervals supported by the system clock.

When your driver calls a routine such as KeSetTimerEx to set a timer, the accuracy of the timer depends on the resolution of the system clock. For more information, see Timer Accuracy.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
IRQL	Any level.

See Also

ExSetTimerResolution

KeSetTimerEx

ExRaiseStatus function

1/25/2019 • 2 minutes to read • Edit Online

The **ExRaiseStatus** routine is called by drivers that supply structured exception handlers to handle particular errors that occur while they are processing I/O requests.

Syntax

```
void ExRaiseStatus(
  NTSTATUS Status
);
```

Parameters

`Status`

Specifies one of the system-defined STATUS_*XXX* values.

Return Value

None

Remarks

Highest-level drivers, particularly file systems, can call **ExRaiseStatus**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlExPassive, HwStorPortProhibitedDDIs

See Also

ExRaiseAccessViolation

ExRaiseDatatypeMisalignment

IoAllocateErrorLogEntry

KeBugCheckEx

ExRegisterCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **ExRegisterCallback** routine registers a given callback routine with a given callback object.

Syntax

```
PVOID ExRegisterCallback(
  PCALLBACK_OBJECT   CallbackObject,
  PCALLBACK_FUNCTION CallbackFunction,
  PVOID              CallbackContext
);
```

Parameters

`CallbackObject`

A pointer to a callback object obtained from the ExCreateCallback routine.

`CallbackFunction`

A pointer to a driver-implemented callback routine, which must be nonpageable. The callback routine must conform to the following prototype:

```
VOID
(*PCALLBACK_FUNCTION ) (
    IN PVOID CallbackContext,
    IN PVOID Argument1,
    IN PVOID Argument2
    );
```

The callback routine parameters are as follows:

CallbackContext

A pointer to a driver-supplied context area as specified in the *CallbackContext* parameter of **ExRegisterCallback**.

Argument1

A pointer to a parameter defined by the callback object.

Argument2

A pointer to a parameter defined by the callback object.

`CallbackContext`

A pointer to a caller-defined structure of data items to be passed as the context parameter of the callback routine each time it is called. Typically the context is part of the caller's device object extension.

Return Value

ExRegisterCallback returns a pointer to a callback registration handle that should be treated as opaque and reserved for system use. This pointer is **NULL** if **ExRegisterCallback** completes with an error.

Remarks

A driver calls **ExRegisterCallback** to register a callback routine with a specified callback object.

If the object allows only one registered callback routine, and such a routine is already registered, **ExRegisterCallback** returns **NULL**.

Callers of **ExRegisterCallback** must save the returned pointer for use later in a call to ExUnregisterCallback. The pointer is required when removing the callback routine from the list of registered callback routines for the callback object.

The meanings of *Argument1* and *Argument2* of the registered callback routine depend on the callback object and are defined by the component that created it. The following are the parameters for the system-defined callback objects:

Starting with Windows Vista, the **\Callback\ProcessorAdd** callback object is available to dynamically track changes in the processor population. The KeRegisterProcessorChangeCallback routine provides similar information, but additionally supports a KE_PROCESSOR_CHANGE_ADD_EXISTING flag that a driver can use to enumerate the processors in the initial multiprocessor system configuration. For drivers that run in Windows Server 2008 and later versions of Windows, use **KeRegisterProcessorChangeCallback** instead of the **\Callback\ProcessorAdd** callback object, if possible.

For more information about callback objects, see Callback Objects.

The operating system calls registered callback routines at the same IRQL at which the driver that created the callback called the ExNotifyCallback routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ExCreateCallback

ExNotifyCallback

ExUnregisterCallback

KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT

KeRegisterProcessorChangeCallback

ExReinitializeResourceLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReinitializeResourceLite** routine reinitializes an existing resource variable.

Syntax

```
NTSTATUS ExReinitializeResourceLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to the caller-supplied resource variable to be reinitialized.

Return Value

ExReinitializeResourceLite returns STATUS_SUCCESS.

Remarks

With a single call to **ExReinitializeResource**, a driver writer can replace three calls: one to **ExDeleteResourceLite**, another to **ExAllocatePool**, and a third to **ExInitializeResourceLite**. As contention for a resource variable increases, memory is dynamically allocated and attached to the resource in order to track this contention. As an optimization, **ExReinitializeResourceLite** retains and zeros this previously allocated memory.

The **ERESOURCE** structure is opaque; that is, the members are reserved for system use.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExConvertExclusiveToSharedLite

ExDeleteResourceLite

ExInitializeResourceLite

ExIsResourceAcquiredExclusiveLite

ExIsResourceAcquiredSharedLite

ExReleaseResourceForThreadLite

ExReInitializeRundownProtection function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReInitializeRundownProtection** routine reinitializes an EX_RUNDOWN_REF structure after the associated object is run down.

Syntax

```
void ExReInitializeRundownProtection(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to an **EX_RUNDOWN_REF** structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

Return Value

None.

Remarks

This routine is called by the driver that owns an object that resides in shared memory and that is accessed by other drivers.

ExReInitializeRundownProtection enables a previously used EX_RUNDOWN_REF structure to be associated with a new object, and initializes run-down protection on this object. After the **ExReInitializeRundownProtection** call, drivers can call ExAcquireRundownProtection to acquire run-down protection on the new object.

An **ExReInitializeRundownProtection** call must be preceded by a call to the ExWaitForRundownProtectionRelease routine that runs down the old object. Between the **ExWaitForRundownProtectionRelease** and **ExReInitializeRundownProtection** calls, the driver might call the ExRundownCompleted routine to indicate that the run down of the old object has completed.

On entry, the status information in the **EX_RUNDOWN_REF** structure must indicate that the old object was run down. Otherwise, **ExReInitializeRundownProtection** bug checks in checked builds.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.

Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExInitializeRundownProtection

ExRundownCompleted

ExWaitForRundownProtectionRelease

ExReleaseResourceForThreadLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseResourceForThreadLite** routine releases the input resource of the indicated thread.

Syntax

```
void ExReleaseResourceForThreadLite(
  PERESOURCE       Resource,
  ERESOURCE_THREAD ResourceThreadId
);
```

Parameters

`Resource`

`ResourceThreadId`

Identifies the thread that originally acquired the resource. If this is not the currently executing thread, the caller must have transferred ownership of the resource by calling the ExSetResourceOwnerPointerEx routine first on the thread that originally acquired the resource.

Return Value

None

Remarks

If this is not the currently executing thread, the caller must have transferred ownership of the resource by calling the ExSetResourceOwnerPointerEx routine first on the thread that originally acquired the resource. This is to ensure that thread A does not get terminated or deleted before thread B has a chance to release the resource.

Unless the caller is running in a system thread, the caller must explicitly disable the delivery of normal kernel APCs before calling this routine. This requirement prevents threads from being suspended while they manipulate or hold a resource. The caller can disable normal kernel APC delivery by calling the KeEnterCriticalRegion routine. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling the KeLeaveCriticalRegion routine. For more information, see Disabling APCs.

This requirement does not apply to calls made to **ExReleaseResourceForThreadLite** from a system thread. A caller running in a system thread does not need to explicitly disable APCs before calling this routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	ExclusiveResourceAccess, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExGetCurrentResourceThread

ExInitializeResourceLite

ExReinitializeResourceLite

ExReleaseResourceLite function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseResourceLite** routine releases a specified executive resource owned by the current thread.

Syntax

```
void ExReleaseResourceLite(
  PERESOURCE Resource
);
```

Parameters

`Resource`

A pointer to an executive resource owned by the current thread.

Return Value

None

Remarks

Normal kernel APC delivery must be disabled before calling this routine. Disable normal kernel APC delivery by calling KeEnterCriticalRegion. Delivery must remain disabled until the resource is released, at which point it can be reenabled by calling KeLeaveCriticalRegion. For more information, see Disabling APCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	ExclusiveResourceAccess, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExAcquireSharedStarveExclusive

ExAcquireSharedWaitForExclusive

ExGetCurrentResourceThread

ExInitializeResourceLite

ExReinitializeResourceLite

ExReleaseResourceLite

ExReleaseRundownProtection function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseRundownProtection** routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtection routine.

Syntax

```
void ExReleaseRundownProtection(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

Return Value

None.

Remarks

A driver that calls ExAcquireRundownProtection to acquire run-time protection for an object access is responsible for calling **ExReleaseRundownProtection** to release the run-time protection when the access is finished.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExInitializeRundownProtection

ExReleaseRundownProtectionEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseRundownProtectionEx** routine releases run-down protection that the caller previously acquired by calling the ExAcquireRundownProtectionEx routine.

Syntax

```
void ExReleaseRundownProtectionEx(
  PEX_RUNDOWN_REF RunRef,
  ULONG           Count
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

`Count`

The amount by which to decrement to the run-down instance count of the object. For more information, see Remarks.

Return Value

None.

Remarks

The *RunRef* parameter points to an EX_RUNDOWN_REF structure that tracks the run-down status of the associated object. This status information includes a count of the instances of run-down protection that are currently in effect on the object. The ExAcquireRundownProtectionEx and **ExReleaseRundownProtectionEx** routines increment and decrement this count by arbitrary amounts. Two related routines, ExAcquireRundownProtection and ExReleaseRundownProtection, increment and decrement this count by one.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows Server 2003.
Target Platform	Universal
Header	wdm.h (include Wdm.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExAcquireRundownProtectionEx

ExInitializeRundownProtection

ExReleaseRundownProtection

ExReleaseSpinLockExclusive function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseSpinLockExclusive** routine releases a spin lock that the caller previously acquired for exclusive access, and restores the IRQL to its original value.

Syntax

```
void ExReleaseSpinLockExclusive(
  PEX_SPIN_LOCK SpinLock,
  KIRQL         OldIrql
);
```

Parameters

`SpinLock`

`OldIrql`

The interrupt request level (IRQL) to restore. Set this parameter to the KIRQL value that was returned by the ExAcquireSpinLockExclusive call that acquired the spin lock.

Return Value

None.

Remarks

This routine must be called only for a spin lock that is owned by the caller.

On entry to this routine, the caller must be running at IRQL = DISPATCH_LEVEL. Before exiting, **ExReleaseSpinLockExclusive** restores the IRQL to the value specified by the *OldIrql* parameter.

The caller should hold the spin lock only briefly before releasing it. For more information, see Introduction to Spin Locks.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1.
Target Platform	Universal
Header	wdm.h
IRQL	DISPATCH_LEVEL (See Remarks.)

See Also

ExAcquireSpinLockExclusive

ExReleaseSpinLockShared function

1/25/2019 • 2 minutes to read • Edit Online

The **ExReleaseSpinLockShared** routine releases ownership of a spin lock that the caller previously acquired for shared access, and restores the IRQL to its original value.

Syntax

```
void ExReleaseSpinLockShared(
  PEX_SPIN_LOCK SpinLock,
  KIRQL         OldIrql
);
```

Parameters

`SpinLock`

`OldIrql`

The interrupt request level (IRQL) to restore. Set this parameter to the KIRQL value that was returned by the ExAcquireSpinLockShared call that acquired the spin lock.

Return Value

None.

Remarks

This routine must be called only for a spin lock that is owned by the caller.

On entry to this routine, the caller must be running at IRQL = DISPATCH_LEVEL. Before exiting, **ExReleaseSpinLockShared** restores the IRQL to the value specified by the *OldIrql* parameter.

The caller should hold the spin lock only briefly before releasing it. For more information, see Introduction to Spin Locks.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1.
Target Platform	Universal
Header	wdm.h
IRQL	DISPATCH_LEVEL (See Remarks.)

See Also

ExAcquireSpinLockShared

ExRundownCompleted function

1/25/2019 • 2 minutes to read • Edit Online

The **ExRundownCompleted** routine updates the run-down status of a shared object to indicate that the run down of the object has completed.

Syntax

```
void ExRundownCompleted(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

Return Value

None.

Remarks

This routine is called by the driver that owns an object that resides in shared memory and that is accessed by other drivers. The **ExRundownCompleted** call must be preceded by a call to the ExWaitForRundownProtectionRelease routine that runs down the object.

After an **ExRundownCompleted** call, additional calls to **ExWaitForRundownProtectionRelease** to run down the associated object return immediately without waiting. However, calls to ExAcquireRundownProtection to acquire run-down protection on the associated object fail (return **FALSE**).

After an **ExRundownCompleted** call, the driver can call the ExReInitializeRundownProtection routine to associate the EX_RUNDOWN_REF structure with a new object and to initialize run-down protection on this object. After the **ExReInitializeRundownProtection** call, drivers can call **ExAcquireRundownProtection** to acquire run-down protection on the new object.

On entry, the status information in the **EX_RUNDOWN_REF** structure must indicate that the object that was associated with this structure was run down. Otherwise, **ExReInitializeRundownProtection** bug checks in checked builds.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.

Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

EX_RUNDOWN_REF

ExInitializeRundownProtection

ExSetFirmwareEnvironmentVariable function

1/25/2019 • 2 minutes to read • Edit Online

The **ExSetFirmwareEnvironmentVariable** routine sets the value of the specified system firmware environment variable.

Syntax

```
NTSTATUS ExSetFirmwareEnvironmentVariable(
  PUNICODE_STRING VariableName,
  LPGUID          VendorGuid,
  PVOID           Value,
  ULONG           ValueLength,
  ULONG           Attributes
);
```

Parameters

`VariableName`

A pointer to a UNICODE_STRING structure that contains the name of the specified environment variable.

`VendorGuid`

A pointer to a GUID that identifies the vendor associated with the specified environment variable. Environment variables are grouped into namespaces based on their vendor GUIDs. Some hardware platforms might not support vendor GUIDs. On these platforms, all variables are grouped into one, common namespace, and the *VendorGuid* parameter is ignored.

`Value`

A pointer to a caller-allocated buffer that contains the data value to write to the specified environment variable.

`ValueLength`

The size, in bytes, of the data value contained in the *Value* buffer.

`Attributes`

The attributes to assign to the specified environment variable. The VARIABLE_ATTRIBUTE_NON_VOLATILE attribute bit must be set or this call will fail. For more information about the attribute bits that are defined for this parameter, see Remarks in ExGetFirmwareEnvironmentVariable.

Return Value

ExSetFirmwareEnvironmentVariable returns STATUS_SUCCESS if it is successful. Possible return values include the following error status codes.

RETURN CODE	DESCRIPTION

STATUS_INSUFFICIENT_RESOURCES	Available system resources are insufficient to complete the requested operation.
STATUS_INVALID_PARAMETER	One of the parameters is not valid.
STATUS_NOT_IMPLEMENTED	This routine is not supported on this platform.
STATUS_UNSUCCESSFUL	The firmware returned an unrecognized error.

Remarks

The caller requires the system environment privilege (SE_SYSTEM_ENVIRONMENT_PRIVILEGE) to use this routine.

System firmware environment variables contain data values that are passed between the boot firmware environment implemented in the hardware platform and the operating-system loaders and other software that runs in the firmware environment. For more information, see Remarks in ExGetFirmwareEnvironmentVariable.

If you create a backup datastore, you can use the ExGetFirmwareEnvironmentVariable routine to save all the boot settings for the platform. Later, you can use **ExSetFirmwareEnvironmentVariable** to restore these settings if needed.

ExSetFirmwareEnvironmentVariable is the kernel-mode equivalent of the Win32 SetFirmwareEnvironmentVariable function.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ExGetFirmwareEnvironmentVariable

SetFirmwareEnvironmentVariable

UNICODE_STRING

ExSetResourceOwnerPointer function

1/25/2019 • 2 minutes to read • Edit Online

The **ExSetResourceOwnerPointer** routine sets the owner thread pointer for an executive resource.

Syntax

```
void ExSetResourceOwnerPointer(
  PERESOURCE Resource,
  PVOID      OwnerPointer
);
```

Parameters

`Resource`

A pointer to an executive resource owned by the current thread.

`OwnerPointer`

A pointer to an owner thread pointer of type ERESOURCE_THREAD (for additional requirements, see the following Remarks section).

Return Value

None

Remarks

ExSetResourceOwnerPointer, used in conjunction with **ExReleaseResourceForThreadLite**, provides a means for one thread (acting as an resource manager thread) to acquire and release resources for use by another thread (acting as a resource user thread).

After calling **ExSetResourceOwnerPointer** for a specific resource, the only other routine that can be called for that resource is **ExReleaseResourceForThreadLite**.

The resource manager thread acquires ownership of the resource and passes ownership to the user thread by calling **ExSetResourceOwnerPointer**. The caller must allocate the memory for the ERESOURCE_THREAD value pointed to by *OwnerPointer* in system memory, and this memory must remain allocated until **ExReleaseResourceForThreadLite** returns. The caller must also set the two low-order bits of the ERESOURCE_THREAD value pointed to by *OwnerPointer* to one — this encoding is used internally by the resource services to distinguish between owner and thread addresses.

When the user thread is done with the resource, the resource manager thread releases the user thread's ownership of the resource by calling **ExReleaseResourceForThreadLite**. The *ResourceThreadId* input parameter is set to the value of the *OwnerPointer* parameter used in the previous call to **ExSetResourceOwnerPointer** that gave the worker thread ownership of the resource.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExReleaseResourceForThreadLite

ExSetResourceOwnerPointerEx function

1/25/2019 • 3 minutes to read • Edit Online

The **ExSetResourceOwnerPointerEx** routine transfers the ownership of an executive resource from the calling thread to an owner pointer, which is a system address that identifies the resource owner.

Syntax

```
void ExSetResourceOwnerPointerEx(
  PERESOURCE Resource,
  PVOID      OwnerPointer,
  ULONG      Flags
);
```

Parameters

`Resource`

A pointer to an executive resource (an opaque ERESOURCE structure) that is owned by the calling thread. The caller previously allocated this structure and initialized it by calling the ExInitializeResourceLite routine. For more information, see the following Remarks section.

`OwnerPointer`

A pointer value that identifies the resource owner. This parameter value is nominally a pointer to a value of type ERESOURCE_THREAD but is cast to type PVOID. The caller must set the two least significant bits of this pointer value to ones. For more information, see the following Remarks section.

`Flags`

A set of flags that can modify the operation of this routine. Set this parameter either to zero or to the following flag value:

- FLAG_OWNER_POINTER_IS_THREAD

If *Flags* = FLAG_OWNER_POINTER_IS_THREAD, *OwnerPointer* must point to a value of type ERESOURCE_THREAD, which is a pointer to an opaque thread object. If *Flags* = 0, *OwnerPointer* must point to a storage object in system memory that remains allocated until the resource is released. For more information, see the following Remarks section.

Return Value

None

Remarks

This routine is available in Windows 7 and later versions of the Windows operating system.

ExSetResourceOwnerPointerEx enables a thread, which acts as a resource manager, to transfer ownership of an executive resource, which the thread has already acquired, to another thread, which then uses the resource. The executive resource is represented by an ERESOURCE structure.

The caller must set the two least significant bits of the *OwnerPointer* parameter value to ones. The operating system uses these bits internally to distinguish an owner pointer that has been set by an **ExSetResourceOwnerPointerEx** call from a thread that has acquired a resource by calling the ExAcquireResourceExclusiveLite or ExAcquireResourceSharedLite routine.

The thread that calls **ExSetResourceOwnerPointerEx** to transfer ownership of a resource must be the same thread that previously called the **ExAcquireResource*Xxx*Lite** routine to acquire the resource.

If the resource-manager thread acquired the resource for exclusive access (by calling **ExAcquireResourceExclusiveLite**), the **ExSetResourceOwnerPointerEx** call transfers exclusive-access ownership to the resource-user thread. If the resource-manager thread acquired the resource for shared access (by calling **ExAcquireResourceSharedLite**), the **ExSetResourceOwnerPointerEx** call transfers shared-access ownership to the resource-user thread.

When the resource-user thread no longer needs the resource, the resource should be released. Typically, the resource-user thread releases the resource. To do so, it must call the ExReleaseResourceForThreadLite routine. In this call, the *ThreadId* parameter value must match the value of the *OwnerPointer* parameter (including the two least significant bits) in the previous **ExSetResourceOwnerPointerEx** call that transferred ownership of the resource.

After a **ExSetResourceOwnerPointerEx** call transfers ownership of a resource, the only other ERESOURCE routine that a driver can call for the specified resource is **ExReleaseResourceForThreadLite**. An attempt to call another **ERESOURCE** routine, such as ExReleaseResourceLite, for this resource is an error.

ExSetResourceOwnerPointerEx can operate in two significantly different modes. The mode is specified by the *Flags* parameter. To select the more versatile of the two modes, set *Flags* = FLAG_OWNER_POINTER_IS_THREAD. This *Flags* value informs the operating system that the *OwnerPointer* parameter is a pointer to the thread object of the resource-manager thread. The operating system can, if necessary, use this information to temporarily boost the priority of this thread to avoid priority inversion. To obtain the thread object pointer to use for the *OwnerPointer* value, a driver can call the ExGetCurrentResourceThread routine. Remember to set the two least significant bits of the *OwnerPointer* value to ones.

If *Flags* is zero, the behavior of **ExSetResourceOwnerPointerEx** is the same as that of the legacy ExSetResourceOwnerPointer routine, which is available in Windows 2000 and later versions of the Windows operating system. That is, the *OwnerPointer* parameter points to a storage object that the resource-manager thread allocates in system memory before it calls **ExSetResourceOwnerPointerEx**. In this case, the object type and value are not relevant because **ExSetResourceOwnerPointerEx** does not attempt to access the object itself. Instead, the routine simply uses the memory address of the object to identify the resource owner. The storage for the object must begin at an even four-byte boundary in memory. To prevent two drivers from inadvertently using the same owner pointer at the same time, the resource-manager thread must not release the storage for the object until the resource is released.

For more information about managing executive resources, see Introduction to ERESOURCE Routines.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ERESOURCE

ExAcquireResourceExclusiveLite

ExAcquireResourceSharedLite

ExGetCurrentResourceThread

ExInitializeResourceLite

ExReleaseResourceForThreadLite

ExReleaseResourceLite

ExSetResourceOwnerPointer

ExSetTimer function

1/25/2019 • 3 minutes to read • Edit Online

The **ExSetTimer** routine starts a timer operation and sets the timer to expire at the specified due time.

Syntax

```
BOOLEAN ExSetTimer(
  PEX_TIMER           Timer,
  LONGLONG            DueTime,
  LONGLONG            Period,
  PEXT_SET_PARAMETERS Parameters
);
```

Parameters

`Timer`

A pointer to an EX_TIMER structure. This structure is a timer object that was previously allocated by the ExAllocateTimer routine.

`DueTime`

The absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes. An absolute time is expressed as the amount of time passed, in system time units, since the start of the year 1601.

The *DueTime* parameter for a high-resolution timer must be a relative time (negative value), or the routine bug checks.

`Period`

An optional period for the timer in system time units (100-nanosecond intervals). Must be less than or equal to MAXLONG. For a timer that is one-shot instead of periodic, set *Period* to zero.

`Parameters`

A pointer to an EXT_SET_PARAMETERS structure. The calling driver previously called the ExInitializeSetTimerParameters routine to initialize this structure.

Return Value

This routine returns **TRUE** if it cancels a timer that was pending at the time that the routine was called. Otherwise, the routine returns **FALSE**. For more information, see Remarks.

Remarks

Your driver can call this routine to set a timer to expire at a future time. The driver can then wait for the timer to expire. Or, the driver can implement a callback routine that is called when the timer expires.

After a driver calls **ExSetTimer**, the driver can call a routine such as KeWaitForSingleObject or or

KeWaitForMultipleObjects to wait for the timer to expire. When the timer expires, the operating system signals the timer object.

As an option, the driver can implement an ExTimerCallback callback routine, and supply a pointer to this routine as an input parameter to the ExAllocateTimer routine. When the timer expires, the operating system calls the *ExTimerCallback* routine.

An **ExSetTimer** call implicitly cancels any previously started set-timer operation on the timer object specified by *Timer*. If your driver previously called **ExSetTimer** to set a timer that uses *Timer*, and this timer has not yet expired when **ExSetTimer** is called a second time, the second call cancels the timer from the first call and then starts the new timer. In this case, the second call returns **TRUE**. However, if the timer started by the first call expires before the second call can cancel this timer, the second call starts the new timer and returns **FALSE**.

If the *Period* parameter is nonzero, the timer is periodic. For a periodic timer, the *DueTime* parameter specifies the time of the initial timer expiration, and *Period* specifies the interval between subsequent expirations.

The hardware timer used for the system clock signals interrupt requests at uniform intervals, but the handling of these interrupts might be delayed by interrupt processing for other devices. Thus, for a periodic timer, the delay from a periodic expiration time to the time at which the timer object is signaled or the *ExTimerCallback* routine runs might vary from one period to the next. For more information, see Timer Accuracy.

A periodic timer can expire no more than once per system clock tick. Setting the period of a timer to a value smaller than the interval between system clock ticks does not cause the timer to expire more than once per system clock tick, but might cause the intervals between successive expirations to vary if the system clock rate changes. For example, the operating system might temporarily increase the system clock rate to meet the timing requirements of a high-resolution timer. For more information, see High-Resolution Timers.

For a periodic timer that is not a high-resolution timer, a driver can achieve relatively uniform periodic expirations by setting *Period* to an integer multiple of the default interval between system clock ticks.

To avoid excessive power consumption, a driver should not set the period of a long-running high-resolution timer to a value less than the default interval between system clock ticks. Otherwise, the system clock timer will continuously generate interrupts at the maximum allowed system clock rate.

For more information, see ExXxxTimer Routines and EX_TIMER Objects.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
IRQL	"<= DISPATCH_LEVEL"

See Also

EXT_SET_PARAMETERS

EX_TIMER

ExAllocateTimer

ExInitializeSetTimerParameters

ExTimerCallback

KeWaitForMultipleObjects

KeWaitForSingleObject

ExSetTimerResolution function

1/25/2019 • 2 minutes to read • Edit Online

The **ExSetTimerResolution** routine modifies the frequency at which the system clock interrupts. Use this routine with extreme caution (see the following Remarks section).

Syntax

```
ULONG ExSetTimerResolution(
  ULONG   DesiredTime,
  BOOLEAN SetResolution
);
```

Parameters

`DesiredTime`

Specifies the amount of time that should elapse between each timer interrupt, in 100-nanosecond units. The minimum value is approximately 10,000 (1 millisecond) but can vary slightly by platform. (This parameter is ignored if *SetResolution* is **FALSE**.)

`SetResolution`

If **TRUE**, the call is a request to set the clock interrupt frequency to the value specified by *DesiredTime*. If **FALSE**, the call is a request to restore the clock interrupt frequency to the system's default value, which is platform-specific.

Return Value

ExSetTimerResolution returns the new timer resolution, in 100-nanosecond units.

Remarks

To set the timer resolution, a driver calls this routine passing **TRUE** as the parameter for *SetResolution*. The following rules apply:

- The routine changes the clock interrupt frequency only if the specified *DesiredTime* value is less than the current setting.
- If a driver requests a *DesiredTime* value that is greater than what is currently set, the routine just returns the current setting.
- If a driver requests a *DesiredTime* value that is less than the system clock can support, the routine uses the smallest resolution the system can support, and returns that value.

If you use this routine to change the clock interrupt frequency, your driver must restore the default interrupt frequency, typically by making the following call before being unloaded:

```
ExSetTimerResolution (0, FALSE);
```

(If multiple drivers have attempted to modify the clock interrupt frequency, the system does not restore the default frequency until all of these drivers have called this routine with a *SetResolution* value of **FALSE**.)

Note that the result of changing the clock interrupt frequency is system-wide and can have a severely negative effect on system performance. Also note that higher clock interrupt frequencies can shorten a system's battery life.

During the processing of an IRP_MJ_POWER request, the power manager holds a lock on a resource that **ExSetTimerResolution** must acquire to complete. Consequently, a deadlock will occur if a driver directly or indirectly calls **ExSetTimerResolution** while processing a power request, and then waits for the call to **ExSetTimerResolution** to return before the driver completes the power request. For more information about safely calling **ExSetTimerResolution** while processing a power IRP, see Calling ExSetTimerResolution While Processing a Power IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

KeSetTimerEx

ExSystemTimeToLocalTime function

1/25/2019 • 2 minutes to read • Edit Online

The **ExSystemTimeToLocalTime** routine converts a GMT system time value to the local system time for the current time zone.

Syntax

```
void ExSystemTimeToLocalTime(
  PLARGE_INTEGER SystemTime,
  PLARGE_INTEGER LocalTime
);
```

Parameters

`SystemTime`

A pointer to a variable set to the unbiased, GMT system time.

`LocalTime`

A pointer to the returned value for the current locale.

Return Value

None

Remarks

ExSystemTimeToLocalTime subtracts the time-zone bias from the GMT system time value to compute the corresponding time at the current locale.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ExLocalTimeToSystemTime

EXT_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

An *ExTimerCallback* callback routine runs after an EX_TIMER timer object's time interval expires.

Syntax

```
EXT_CALLBACK ExtCallback;

void ExtCallback(
  PEX_TIMER Timer,
  PVOID Context
)
{...}
```

Parameters

`Timer`

A pointer to an EX_TIMER structure. This structure is a timer object that was previously allocated by the ExAllocateTimer routine.

`Context`

The context value that your driver previously supplied as the *CallbackContext* parameter of the ExAllocateTimer routine.

Return Value

None.

Remarks

As an option, your driver can supply a pointer to an *ExTimerCallback* routine in the *Callback* parameter that your driver passes to the ExAllocateTimer routine. After the timer expires, the operating system calls the *ExTimerCallback* routine.

To start a timer operation, the driver passes the timer object as an input parameter to the ExSetTimer routine. After the timer expires, the operating system schedules the *ExTimerCallback* routine to run.

A timer object is a waitable object. A driver thread can call a routine such as KeWaitForSingleObject or KeWaitForMultipleObjects to wait for the timer to expire. When the timer expires, the operating system signals the timer object.

A timer might be canceled before it expires. The driver can call the ExCancelTimer routine to explicitly cancel a timer. If the driver calls the ExSetTimer routine to start a new timer on a timer object before a previously set timer on this object expires, this call implicitly cancels the previous timer and then starts the new timer.

An *ExTimerCallback* routine can call a routine such as **ExSetTimer** or ExDeleteTimer. If an *ExTimerCallback* routine calls **ExDeleteTimer**, the *Wait* parameter supplied in this call must be **FALSE**. For more information, see Deleting a System-Allocated Timer Object.

In a multiprocessor system, *ExTimerCallback* callbacks for two successive expirations of a periodic timer might

simultaneously run on two different processors.

For more information, see ExXxxTimer Routines and EX_TIMER Objects.

Examples

To define an *ExTimerCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function type helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *ExTimerCallback* callback routine that is named `MyExTimerCallback`, use the EXT_CALLBACK function type, as shown in this code example:

```
EXT_CALLBACK  MyExTimerCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyExTimerCallback(
    PEX_TIMER  Timer,
    PVOID  Context
    )
  {...}
```

The EXT_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the ***Use_decl_annotations*** annotation to your function definition. The ***Use_decl_annotations*** annotation ensures that the annotations that are applied to the EXT_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions Using Function Role Types for WDM Drivers. For more information about ***Use_decl_annotations***, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

See Also

EX_TIMER

ExAllocateTimer

ExCancelTimer

ExSetTimer

KeWaitForMultipleObjects

KeWaitForSingleObject

EXT_DELETE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

An *ExTimerDeleteCallback* callback routine runs when the operating system deletes an EX_TIMER timer object.

Syntax

```
EXT_DELETE_CALLBACK ExtDeleteCallback;

void ExtDeleteCallback(
  PVOID Context
)
{...}
```

Parameters

`Context`

The context value from the *DeleteContext* member of the EXT_DELETE_PARAMETERS structure that your driver previously passed as an input parameter to the ExDeleteTimer routine.

Return Value

None.

Remarks

As an option, your driver can supply a pointer to an *ExTimerDeleteCallback* routine in the *DeleteCallback* member of the EXT_DELETE_PARAMETERS structure that your driver passes as an input parameter to the ExDeleteTimer routine. The *ExTimerDeleteCallback* routine can free any storage or other system resources that the driver might have previously allocated to use with the timer object that is being deleted.

If the driver supplies an *ExTimerDeleteCallback* routine, and the *Wait* parameter in the **ExDeleteRoutine** call is **TRUE**, the *ExTimerDeleteCallback* routine runs before **ExDeleteTimer** returns. Otherwise, the *ExTimerDeleteCallback* routine might run before or after the **ExDeleteTimer** call returns. The *ExTimerDeleteCallback* routine is called only after the timer object has been disabled to prevent further timer operations and any pending timer operation on the timer object is canceled or completed. The timer object (an EX_TIMER structure) that the driver passes as an input parameter to the **ExDeleteTimer** routine might no longer be valid by the time the *ExTimerDeleteCallback* routine runs.

For more information, see ExXxxTimer Routines and EX_TIMER Objects.

Examples

To define an *ExTimerDeleteCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function type helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *ExTimerDeleteCallback* callback routine that is named `MyExTimerDeleteCallback`, use the EXT_DELETE_CALLBACK function type, as shown in this code example:

```
EXT_DELETE_CALLBACK  MyExTimerDeleteCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyExTimerDeleteCallback(
    PVOID  Context
    )
  {...}
```

The EXT_DELETE_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the ***Use_decl_annotations*** annotation to your function definition. The ***Use_decl_annotations*** annotation ensures that the annotations that are applied to the EXT_DELETE_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions Using Function Role Types for WDM Drivers. For more information about ***Use_decl_annotations***, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

See Also

EXT_DELETE_PARAMETERS

EX_TIMER

ExDeleteTimer

ExTryConvertSharedSpinLockExclusive function

1/25/2019 • 2 minutes to read • Edit Online

The **ExTryConvertSharedSpinLockExclusive** routine attempts to convert the access state of a spin lock from *acquired for shared access* to *exclusive access*.

Syntax

```
LOGICAL ExTryConvertSharedSpinLockExclusive(
  PEX_SPIN_LOCK SpinLock
);
```

Parameters

`SpinLock`

A pointer to the spin lock whose access state is to be converted to exclusive access. The caller must already own this spin lock for shared access.

Return Value

ExTryConvertSharedSpinLockExclusive returns TRUE if the conversion succeeds; otherwise, it returns FALSE.

Remarks

If the caller acquired the shared spin lock by calling the ExAcquireSpinLockSharedAtDpcLevel routine, the caller should release the converted spin lock by calling the ExReleaseSpinLockExclusiveFromDpcLevel routine. If the caller acquired the shared spin lock by calling the ExAcquireSpinLockShared routine, the caller should release the converted spin lock by calling the ExReleaseSpinLockExclusive routine, and the *OldIrql* value supplied as an input parameter to this routine should be the KIRQL value returned by **ExAcquireSpinLockShared**.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
IRQL	DISPATCH_LEVEL

See Also

ExAcquireSpinLockShared

ExAcquireSpinLockSharedAtDpcLevel

ExReleaseSpinLockExclusive

ExReleaseSpinLockExclusiveFromDpcLevel

ExUnregisterCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **ExUnregisterCallback** routine removes a callback routine previously registered with a callback object from the list of routines to be called during the notification process.

Syntax

```
void ExUnregisterCallback(
  PVOID CallbackRegistration
);
```

Parameters

`CallbackRegistration`

Specifies the callback routine to unregister. This must be the value returned by **ExRegisterCallback** when the callback was registered.

Return Value

None

Remarks

For more information about callback objects, see Callback Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ExCreateCallback

ExRegisterCallback

ExWaitForRundownProtectionRelease function

1/25/2019 • 2 minutes to read • Edit Online

The **ExWaitForRundownProtectionRelease** routine waits until all drivers that have already been granted run-down protection complete their accesses of the shared object.

Syntax

```
void ExWaitForRundownProtectionRelease(
  PEX_RUNDOWN_REF RunRef
);
```

Parameters

`RunRef`

A pointer to an EX_RUNDOWN_REF structure that was initialized by a previous call to the ExInitializeRundownProtection routine. The run-down protection routines use this structure to track the run-down status of the associated shared object. This structure is opaque to drivers.

Return Value

None.

Remarks

This routine is called by the driver that owns an object that resides in shared memory and that is accessed by other drivers. Before deleting this object, the owning driver must call this routine to wait for any outstanding accesses of the object to complete. After **ExWaitForRundownProtectionRelease** returns, the owning driver can safely delete the object.

After **ExWaitForRundownProtectionRelease** is called, the ExAcquireRundownProtection routine grants no further requests for run-down protection from drivers that are trying to access the shared object. The routine waits to return until all drivers that were previously granted run-down protection finish accessing the object. As each driver finishes, it calls the ExReleaseRundownProtection routine to release the previously acquired run-down protection. When all outstanding accesses are completed, **ExWaitForRundownProtectionRelease** returns and the object can be safely deleted.

If **ExWaitForRundownProtectionRelease** is called when all drivers that were previously granted run-down protection have already finished accessing the shared object, the routine changes the object status to *run down*, and returns immediately, without waiting.

If **ExWaitForRundownProtectionRelease** is called to run down a shared object, but the *RunRef* parameter indicates that this object is already run down, the call has no effect but is not treated as an error.

For more information, see Run-Down Protection.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs, SpNoWait

See Also

EX_RUNDOWN_REF

ExAcquireRundownProtection

ExInitializeRundownProtection

ExReleaseRundownProtection

FIELD_OFFSET macro

1/25/2019 • 2 minutes to read • Edit Online

The **FIELD_OFFSET** macro returns the byte offset of a named field in a known structure type.

Syntax

```
void FIELD_OFFSET(
   type,
   field
);
```

Parameters

`type`

Specifies the name of a known structure type containing *Field*.

`field`

Specifies the name of a field in a structure of type *Type*.

Return Value

None

Remarks

Used by device driver writers to symbolically determine the offset of a known field in a known structure type.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Miniport.h, Minitape.h, Scsi.h, Storport.h)
IRQL	Any level

See Also

CONTAINING_RECORD

FirstEntrySList function

1/25/2019 • 2 minutes to read • Edit Online

The **FirstEntrySList** routine returns the first entry in a sequenced singly linked list.

Syntax

```
PSLIST_ENTRY FirstEntrySList(
  PSLIST_HEADER SListHead
);
```

Parameters

`SListHead`

Pointer to the SLIST_HEADER structure that serves as the header for the sequenced singly linked list.

Return Value

FirstEntrySList returns a pointer to the first SLIST_ENTRY structure on the list. If the list is empty, the routine returns **NULL**.

Remarks

Unlike other sequenced singly linked list routines, the **FirstEntrySList** routine is not atomic. For more information about sequenced singly linked lists, see Sequenced Singly Linked Lists. Callers of this routine can run at any level. If called at IRQL >= DISPATCH_LEVEL, the storage for **ListHead** must be resident.

Requirements

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

SLIST_ENTRY

SLIST_HEADER

FREE_FUNCTION_EX callback function

1/25/2019 • 2 minutes to read • Edit Online

The *LookasideListFreeEx* routine frees the storage for a lookaside-list entry when a client tries to insert the entry into a lookaside list that is full.

Syntax

```
FREE_FUNCTION_EX FreeFunctionEx;

void FreeFunctionEx(
  PVOID Buffer,
  PLOOKASIDE_LIST_EX Lookaside
)
{...}
```

Parameters

`Buffer`

A pointer to the lookaside-list entry that is to be freed.

`Lookaside`

A pointer to a LOOKASIDE_LIST_EX structure that describes the lookaside list. This structure was previously initialized by the ExInitializeLookasideListEx routine.

Return Value

None

Remarks

A driver that creates a lookaside list can implement a *LookasideListFreeEx* routine to free previously allocated buffers when the list becomes full and can contain no more buffers. A buffer that is not in use is stored as an entry in the list. All entries in a lookaside list are buffers of a uniform size, which the driver specifies when the list is initialized.

The driver supplies a pointer to a custom *LookasideListFreeEx* routine as an input parameter in the **ExInitializeLookasideListEx** call that initializes the lookaside list. If the driver sets this parameter to **NULL**, the lookaside list uses a default deallocation routine instead.

A driver calls the ExFreeToLookasideListEx routine to insert a previously allocated entry into a lookaside list. If the list is full (that is, if the list already contains the system-determined maximum number of entries), **ExFreeToLookasideListEx** calls *LookasideListFreeEx* to free the storage for the entry.

The *LookasideListFreeEx* routine can use the *Lookaside* parameter to access private context data that the driver has associated with the lookaside list. For more information, see the code example in ExInitializeLookasideListEx.

For more information about lookaside lists, see Using Lookaside Lists.

The *LookasideListFreeEx* routine is called at the same IRQL as the call to **ExFreeToLookasideListEx** that frees the entry. For a call to free an entry that resides in paged memory, the caller must be running IRQL <= APC_LEVEL.

For a call to free an entry that resides in nonpaged memory, the caller must be running IRQL <= DISPATCH_LEVEL.

Examples

To define a *LookasideListFreeEx* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *LookasideListFreeEx* callback routine that is named `MyLookasideListFreeEx`, use the FREE_FUNCTION_EX type as shown in this code example:

```
FREE_FUNCTION_EX MyLookasideListFreeEx;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyLookasideListFreeEx(
    PVOID  Buffer,
    PLOOKASIDE_LIST_EX  Lookaside
    )
  {
      // Function body
  }
```

The FREE_FUNCTION_EX function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the FREE_FUNCTION_EX function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows Vista and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	See Remarks section.

See Also

ExFreeToLookasideListEx

ExInitializeLookasideListEx

LOOKASIDE_LIST_EX

GET_D3COLD_CAPABILITY callback function

1/25/2019 • 2 minutes to read • Edit Online

The *GetBusDriverD3ColdSupport* routine enables the driver for a device to query whether the enumerating bus driver supports the D3cold device power state.

Syntax

```
GET_D3COLD_CAPABILITY GetD3coldCapability;

NTSTATUS GetD3coldCapability(
  PVOID Context,
  PBOOLEAN D3ColdSupported
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller sets this parameter to the value of the **Context** member of the D3COLD_SUPPORT_INTERFACE structure for the interface.

`D3ColdSupported`

A pointer to a BOOLEAN variable to which the routine writes a value to indicate whether the bus driver supports the D3cold. If this value is **TRUE**, the bus driver supports D3cold. If **FALSE**, the bus driver does not support D3cold. If the call fails, the routine returns an error status code and does not write anything to this variable.

Return Value

The *GetBusDriverD3ColdSupport* routine returns STATUS_SUCCESS if it is successful. Otherwise, it returns an appropriate error status code.

Remarks

The driver for the device calls the version of this routine that is implemented by the Windows ACPI driver, Acpi.sys. This routine checks the parent bus driver for the device to determine whether this bus driver supports the D3cold power state.

For example, starting with Windows 8, Microsoft supplies an inbox USB 3.0 eXtensible Host Controller Interface (xHCI) driver that supports D3cold. Some third-party hardware vendors supply Windows drivers for their xHCI controllers, but these drivers might not support D3cold. The driver for a USB 3.0 device can call the *GetBusDriverD3ColdSupport* routine to determine whether the parent xHCI controller driver supports D3cold.

A bus driver supports D3cold if all of the following are true:

- The bus driver implements the GUID_D3COLD_SUPPORT_INTERFACE driver interface.
- The bus driver implements the *GetBusDriverD3ColdSupport* routine in this interface.
- The output value from the *GetBusDriverD3ColdSupport* routine indicates that the bus driver supports D3cold.

The driver for a device can call the GetIdleWakeInfo routine to determine whether the underlying bus drivers and ACPI

system firmware support D3cold for the device. If this call fails and returns an error status code, the device driver can call the *GetBusDriverD3ColdSupport* routine to determine whether the failure is caused by lack of D3cold support by the parent bus driver.

A device on a bus can make a transition to the D3cold substate only if the bus driver supports this transition. If the bus driver does not support D3cold, the device never enters D3cold, even if the function driver for the device calls the SetD3ColdSupport routine to enable the transition to D3cold. In this case, *SetD3ColdSupport* calls have no effect, but are harmless.

For this reason, most device drivers never need to call the *GetBusDriverD3ColdSupport* routine.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	PASSIVE_LEVEL

See Also

D3COLD_SUPPORT_INTERFACE

GUID_D3COLD_SUPPORT_INTERFACE

GetIdleWakeInfo

SetD3ColdSupport

GET_D3COLD_LAST_TRANSITION_STATUS callback function

1/25/2019 • 2 minutes to read • Edit Online

The *GetLastTransitionStatus* routine enables the driver for a device to query whether the most recent transition to the D3hot substate was followed by a transition to the D3cold substate.

Syntax

```
GET_D3COLD_LAST_TRANSITION_STATUS GetD3coldLastTransitionStatus;

void GetD3coldLastTransitionStatus(
  PVOID Context,
  PD3COLD_LAST_TRANSITION_STATUS LastTransitionStatus
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller sets this parameter to the value of the **Context** member of the D3COLD_SUPPORT_INTERFACE structure for the interface.

`LastTransitionStatus`

A pointer to a variable into which the routine writes one of the following D3COLD_LAST_TRANSITION_STATUS enumeration values:

- **LastDStateTransitionStatusUnknown**
- **LastDStateTransitionD3hot**
- **LastDStateTransitionD3cold**

The **LastDStateTransitionD3cold** value indicates that the most recent transition to D3hot was followed by a transition to D3cold. For more information, see Remarks.

Return Value

None.

Remarks

This routine tries to get the information needed to answer the caller's query from the parent bus driver and platform firmware. If this information is not available, the routine writes the value **LastDStateTransitionStatusUnknown** to the location pointed to by *the LastTransitionStatus* parameter.

For more information, see D3COLD_LAST_TRANSITION_STATUS.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	PASSIVE_LEVEL

See Also

D3COLD_LAST_TRANSITION_STATUS

D3COLD_SUPPORT_INTERFACE

GET_DMA_ADAPTER callback function

1/25/2019 • 2 minutes to read • Edit Online

The *GetDmaAdapter* routine returns a DMA_ADAPTER structure for the target device.

Syntax

```
GET_DMA_ADAPTER GetDmaAdapter;

_DMA_ADAPTER * GetDmaAdapter(
  PVOID Context,
  _DEVICE_DESCRIPTION *DeviceDescriptor,
  PULONG NumberOfMapRegisters
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the BUS_INTERFACE_STANDARD structure for the interface.

`DeviceDescriptor`

A pointer to a DEVICE_DESCRIPTION structure that describes the attributes of the physical device.

`NumberOfMapRegisters`

A pointer to, on output, the maximum number of map registers that the driver can allocate for any DMA transfer operation. The caller must allocate a **LONG** variable to receive this data.

Return Value

The *GetDmaAdapter* routine returns a pointer to a DMA_ADAPTER structure when successful. If an adapter structure cannot be allocated, the routine returns **NULL**.

Remarks

If the driver is executing at IRQL = PASSIVE_LEVEL, it should obtain a device's DMA adapter object by calling the IoGetDmaAdapter function. **IoGetDmaAdapter** detects whether the bus driver supports the BUS_INTERFACE_STANDARD interface; if it does, **IoGetDmaAdapter** calls the routine pointed to by the *GetDmaAdapter* member of this interface to obtain the adapter object. Otherwise, **IoGetDmaAdapter** calls an equivalent legacy routine.

However, if a driver must obtain an adapter object while running at IRQL ≥ DISPATCH_LEVEL, it cannot do so with the IoGetDmaAdapter function. In such a case, the driver must query for the BUS_INTERFACE_STANDARD interface while still at IRQL = PASSIVE_LEVEL by issuing an IRP_MN_QUERY_INTERFACE request.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Ntddk.h, Wdm.h)
IRQL	PASSIVE_LEVEL

See Also

BUS_INTERFACE_STANDARD

DEVICE_DESCRIPTION

DMA_ADAPTER

IoGetDmaAdapter

GET_IDLE_WAKE_INFO callback function

1/25/2019 • 5 minutes to read • Edit Online

The *GetIdleWakeInfo* routine enables the driver for a device to discover the device power states from which the device can signal a wake event.

Syntax

```
GET_IDLE_WAKE_INFO GetIdleWakeInfo;

NTSTATUS GetIdleWakeInfo(
  PVOID Context,
  SYSTEM_POWER_STATE SystemPowerState,
  PDEVICE_WAKE_DEPTH DeepestWakeableDstate
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller sets this parameter to the value of the **Context** member of the D3COLD_SUPPORT_INTERFACE structure for the interface.

`SystemPowerState`

System power state. Set this parameter to one of the following SYSTEM_POWER_STATE enumeration values:

- **PowerSystemWorking**
- **PowerSystemSleeping1**
- **PowerSystemSleeping2**
- **PowerSystemSleeping3**
- **PowerSystemHibernate**

These values represent system power states S0 (system working state) through S4. For the *SystemPowerState* value supplied by the caller, the routine determines the deepest device power state from which the device can issue a wake signal.

`DeepestWakeableDstate`

Deepest wakeable Dx state. This parameter is a pointer to a DEVICE_WAKE_DEPTH variable. If the call is successful, the routine writes one of the following enumeration values to this variable:

- **DeviceWakeDepthNotWakeable**
- **DeviceWakeDepthD0**
- **DeviceWakeDepthD1**
- **DeviceWakeDepthD2**
- **DeviceWakeDepthD3hot**
- **DeviceWakeDepthD3cold**

A value in the range **DeviceWakeDepthD0** to **DeviceWakeDepthD3cold** indicates the lowest-powered Dx state from which the device can send a wake signal when the computer is in the system power state specified by the

SystemPowerState parameter. **DeviceWakeDepthNotWakeable** indicates that there is no device power state from which the device can send a wake signal when the computer is in the system power state specified by *SystemPowerState*.

If the routine cannot determine the deepest wakeable device state (perhaps because the platform firmware does not contain this information), the call fails and the routine returns an error status code. If a *GetIdleWakeInfo* call fails for any *SystemPowerState* parameter value in the range **PowerSystemWorking** to **PowerSystemHibernate**, it will fail for all such values.

Return Value

The *GetIdleWakeInfo* routine returns STATUS_SUCCESS if it successfully retrieves the deepest wakeable device state. Otherwise, it returns an appropriate error status code.

Remarks

For the system power state specified by the caller, this routine tries to determine the lowest-powered device power state from which the device can signal a wake event to the processor. If successful, the routine writes the device power state to the location pointed to by the *DeepestWakeableDstate* parameter, and returns STATUS_SUCCESS. Or, if the routine determines that the device cannot signal a wake event from any device power state, the routine writes the value **DeviceWakeDepthNotWakeable** to this location, and returns STATUS_SUCCESS.

The driver for a device uses the information supplied by the *GetIdleWakeInfo* routine to determine the conditions under which the device can signal a wake event. A device that needs to be able to signal certain wake events should not enter a device power state from which it cannot signal these events. For some types of devices, the wake signals that the device should send when the computer is in the S0 (working) system power state differ from those that the device should send when the computer appears to be off.

For example, when a card is inserted into a PCI Express card slot, and the PCI Express hot-plug controller device for the slot is in the D0 device power state, this device signals an interrupt to the processor. However, if the controller device is in a low-power Dx state when the card is inserted, the system power state of the computer might determine whether this device should signal a wake event to the processor. Ideally, the controller device should behave as follows:

- If the computer is in the S0 (working) system power state, the device should signal a wake event to the processor.
- If the computer is sleeping (in a low-power Sx state), the device should not signal a wake event.

Some older devices might not support this ideal behavior. If the PCI Express hot-plug controller device in this example can signal a wake event only when the computer is in the S3 state, the driver (which, in this case, is the inbox Pci.sys driver) for the controller should keep the controller in D0 when the computer is in S0 (and is not preparing to enter a sleeping state).

The driver in this example can call the *GetIdleWakeInfo* routine to determine whether the hot-plug controller device should leave the D0 state while the computer is in S0. For this call, the driver sets *SystemPowerState* = **PowerSystemWorking**. This device should not leave the D0 state (when the computer is in S0) in either of the following cases:

- The *GetIdleWakeInfo* call sets the output parameter to either **DeviceWakeDepthNotWakeable** or **DeviceWakeDepthD0** and returns STATUS_SUCCESS.
- The *GetIdleWakeInfo* call fails and returns an error status code.

The driver should interpret either of these results to mean that the device cannot signal a wake event when the computer is in S0. Based on this information, the driver should keep the device in D0 until the computer prepares to exit S0.

The drivers for most devices can treat an output value of **DeviceWakeDepthD0** the same as **DeviceWakeDepthNotWakeable**. Only a few drivers might have a reason to arm a wake signal when their devices are in D0. These are drivers for simple devices that monitor external events that trigger wake signals regardless of the whether the devices are in D0 or low-power Dx states. An example of such a device is a power button or a sleep button on a computer.

The *GetIdleWakeInfo* routine queries the underlying bus driver and ACPI system firmware to determine the lowest device power state from which the device can signal a wake event. If the bus driver and firmware cannot supply this information, the routine fails and returns an error status code.

The DEVICE_CAPABILITIES structure includes a **DeviceWake** member that provides information similar to that available from the *GetIdleWakeInfo* routine. However, the information in the **DeviceWake** member applies only to the system low-power states S1 to S4. For some devices, the information in the **DeviceWake** member might also apply to the S0 system power state, but drivers should not rely on such behavior. Only the *GetIdleWakeInfo* routine reliably reports the ability of a device to signal a wake event if the computer is in S0.

An inline helper function, **MapWakeDepthToDstate**, is provided to convert the DEVICE_WAKE_DEPTH output value from the *GetIdleWakeInfo* routine to a DEVICE_POWER_STATE value that can be used as an input parameter by the PoRequestPowerIrp routine.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	PASSIVE_LEVEL

See Also

D3COLD_SUPPORT_INTERFACE

DEVICE_CAPABILITIES

DEVICE_POWER_STATE

DEVICE_WAKE_DEPTH

PoRequestPowerIrp

SYSTEM_POWER_STATE

GET_SDEV_IDENTIFIER callback function

1/25/2019 • 2 minutes to read • Edit Online

This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.

Syntax

```
GET_SDEV_IDENTIFIER GetSdevIdentifier;

ULONGLONG GetSdevIdentifier(
  PVOID InterfaceContext
)
{...}
```

Parameters

`InterfaceContext`

Return Value

None

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h (include Wdm.h)

GET_SET_DEVICE_DATA callback function

1/25/2019 • 2 minutes to read • Edit Online

The *GetBusData* routine reads data from the device's configuration space.

Syntax

```
GET_SET_DEVICE_DATA GetSetDeviceData;

ULONG GetSetDeviceData(
  PVOID Context,
  ULONG DataType,
  PVOID Buffer,
  ULONG Offset,
  ULONG Length
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the BUS_INTERFACE_STANDARD structure for the interface.

`DataType`

The type of bus data to read. For more information, see the discussion of the *WhichSpace* parameter in IRP_MN_READ_CONFIG.

`Buffer`

A pointer to the buffer that holds the configuration information to be read from the device's configuration space.

`Offset`

The offset into the bus-specific device data where this read operation begins. For example, if the *DataType* parameter is **PCI_WHICHSPACE_CONFIG**, the *Offset* parameter value is the offset into PCI Configuration Space.

`Length`

The length, in bytes, of data to read.

Return Value

The *GetBusData* routine returns the length, in bytes, of the configuration data that is read after a successful read operation. If the read operation is unsuccessful, a value of zero is returned.

Remarks

Depending on the value of the *DataType* parameter, the *GetBusData* routine can be called only from the interrupt request levels (IRQLs) that are defined in the following table.

***DATATYPE* VALUE**	**IRQL**

PCI_WHICHSPACE_CONFIG	<= DIRQL
PCI_WHICHSPACE_ROM	<= APC_LEVEL
PCCARD_COMMON_MEMORY **PCCARD_COMMON_MEMORY_INDIRECT**	<= DISPATCH_LEVEL
PCCARD_ATTRIBUTE_MEMORY **PCCARD_ATTRIBUTE_MEMORY_INDIRECT**	<= DISPATCH_LEVEL
PCCARD_PCI_CONFIGURATION_SPACE	<= DIRQL

The following remarks apply to drivers for PCI Express (PCIe) virtual functions (VFs) on devices that support the single root I/O virtualization (SR-IOV) interface:

- Drivers for PCIe VFs should call the *GetBusData* routine at IRQL <= APC_LEVEL.
- In order to read PCI Configuration data for the VF at IRQL = DISPATCH_LEVEL, the driver must issue I/O requests of IRP_MN_READ_CONFIG.

Requirements

Target Platform	Desktop
Header	wdm.h (include Ntddk.h)
IRQL	See Remarks section.

See Also

BUS_INTERFACE_STANDARD

IRP_MN_READ_CONFIG

InitializeListHead function

1/25/2019 • 2 minutes to read • Edit Online

The **InitializeListHead** routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.

Syntax

```
void InitializeListHead(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to a LIST_ENTRY structure that serves as the list header.

Return Value

None

Remarks

The **InitializeListHead** routine sets the **Flink** and **Blink** members of *ListHead* to point to *ListHead*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InitializeListHead** can be running at any IRQL. If **InitializeListHead** is called at IRQL >= DISPATCH_LEVEL the storage for *ListHead* must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedInsertHeadList

ExInterlockedInsertTailList

ExInterlockedPopEntryList

ExInterlockedPushEntryList

ExInterlockedRemoveHeadList

InsertHeadList

InsertTailList

IsListEmpty

KeInitializeSpinLock

PopEntryList

PushEntryList

RemoveEntryList

RemoveHeadList

RemoveTailList

InitializeSListHead function

1/25/2019 • 2 minutes to read • Edit Online

The **InitializeSListHead** routine (or ExInitializeSListHead) initializes an SLIST_HEADER structure that represents the head of a sequenced singly linked list.

Syntax

```
void InitializeSListHead(
  PSLIST_HEADER SListHead
);
```

Parameters

`SListHead`

A pointer to the SLIST_HEADER structure to initialize. The structure must be 16-byte aligned on 64-bit platforms.

Return Value

None

Remarks

For more information about using this routine to implement a sequenced singly linked list, see Singly and Doubly Linked Lists.

Callers of **InitializeSListHead or ExInitializeSListHead** can be running at any IRQL. If **ExInitializeSListHead** is called at IRQL >= DISPATCH_LEVEL the storage for *SListHead* must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

See Also

ExInitializeNPagedLookasideList

ExInitializePagedLookasideList

ExInterlockedInsertTailList

ExInterlockedPopEntrySList

ExInterlockedPushEntrySList

ExQueryDepthSList

KeInitializeSpinLock

InsertHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **InsertHeadList** routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.

Syntax

```
void InsertHeadList(
  PLIST_ENTRY                 ListHead,
  __drv_aliasesMem PLIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that represents the head of the list.

`Entry`

Pointer to a LIST_ENTRY structure that represents the entry to be inserted into the list.

Return Value

None

Remarks

InsertHeadList updates *ListHead*->**Flink** to point to *Entry*. It updates *Entry*->**Flink** to point to the old first entry in the list, and sets *Entry*->**Blink** to *ListHead*. The **Blink** field of the original first entry is also updated to point to *Entry*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertHeadList** can be running at any IRQL. If **InsertHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

DDI compliance rules	IoAllocateFree, IoReuseIrp, MarkingQueuedIrps, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

ExInterlockedInsertHeadList

InitializeListHead

InsertTailList

IsListEmpty

RemoveHeadList

RemoveTailList

InsertTailList function

1/25/2019 • 2 minutes to read • Edit Online

The **InsertTailList** routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.

Syntax

```
void InsertTailList(
  PLIST_ENTRY                 ListHead,
  __drv_aliasesMem PLIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that represents the head of the list.

`Entry`

Pointer to a LIST_ENTRY structure that represents the entry to be inserted in the list.

Return Value

None

Remarks

InsertTailList updates *ListHead*->**Blink** to point to *Entry*. It updates *Entry*->**Blink** to point to the old last entry in the list, and sets *Entry*->**Flink** to *ListHead*. The **Flink** of the previous last entry is updated to point to *Entry* as well.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertTailList** can be running at any IRQL. If **InsertTailList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedInsertTailList

InitializeListHead

InsertHeadList

IsListEmpty

RemoveHeadList

RemoveTailList

InterlockedAnd function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedAnd** macro atomically computes a bitwise AND operation.

Syntax

```
LONG InterlockedAnd(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be ANDed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be ANDed with the variable that is pointed to by *Destination*.

Return Value

InterlockedAnd returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedAnd atomically computes ***Destination*&=*Value***.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedOr

InterlockedXor

InterlockedCompareExchange function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedCompareExchange** routine performs an atomic operation that compares the input value pointed to by *Destination* with the value of *Comperand*.

Syntax

```
LONG CDECL_NON_WVMPURE InterlockedCompareExchange(
  LONG volatile *Destination,
  LONG          ExChange,
  LONG          Comperand
);
```

Parameters

`Destination`

A pointer to the input value that is compared with the value of *Comperand*.

`ExChange`

Specifies the output value pointed to by *Destination* if the input value pointed to by *Destination* equals the value of *Comperand*.

`Comperand`

Specifies the value that is compared with the input value pointed to by *Destination*.

Return Value

InterlockedCompareExchange returns the original value of **Destination*.

Remarks

If *Comperand* is equal to **Destination*, then **Destination* is set to equal *Exchange*. Otherwise, **Destination* is unchanged.

InterlockedCompareExchange provides a fast, atomic way to synchronize the testing and updating of a variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comperand*, the output value of *Destination* is set to the value of *Exchange*.

InterlockedCompareExchange is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchange** is atomic only with respect to other **Interlocked*Xxx*** calls. It does not use a spin lock and can be safely used on pageable data.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedCompareExchange64

InterlockedCompareExchangePointer

InterlockedDecrement

InterlockedExchange

InterlockedExchangePointer

InterlockedIncrement

InterlockedCompareExchangePointer function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedCompareExchangePointer** routine performs an atomic operation that compares the input pointer value pointed to by *Destination* with the pointer value *Comperand*.

Syntax

```
PVOID InterlockedCompareExchangePointer(
  PVOID volatile *Destination,
  PVOID          Exchange,
  PVOID          Comperand
);
```

Parameters

`Destination`

A pointer to a PVOID value. If (*Destination) = Comperand, then the routine sets (Destination*) to *Exchange*.

`Exchange`

Specifies the PVOID value to set (**Destination*) to.

`Comperand`

Return Value

InterlockedCompareExchangePointer returns the original value of the pointer at **Destination* (that is, the value of this pointer at entry to the routine).

Remarks

If *Comperand* is equal to **Destination*, then **Destination* is set to equal *Exchange*. Otherwise, **Destination* is unchanged.

InterlockedCompareExchangePointer provides a fast, atomic way to synchronize the testing and updating of a pointer variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comperand*, the value pointed to by *Destination* is set to the value of *Exchange*.

InterlockedCompareExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchangePointer** is atomic only with respect to other **Interlocked*Xxx*** calls. It does not use a spin lock and can be safely used on pageable data.

The **InterlockedCompareExchangePointer** routine is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedCompareExchange

InterlockedExchange

InterlockedExchangePointer

InterlockedDecrement function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedDecrement** routine decrements a caller-supplied variable of type LONG as an atomic operation.

Syntax

```
LONG CDECL_NON_WVMPURE InterlockedDecrement(
  LONG volatile *Addend
);
```

Parameters

`Addend`

A pointer to a variable to be decremented.

Return Value

InterlockedDecrement returns the decremented value.

Remarks

InterlockedDecrement should be used instead of **ExInterlockedDecrementLong** because it is both more efficient and faster.

InterlockedDecrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedDecrement is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedExchange

InterlockedIncrement

InterlockedExchange function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchange** routine sets an integer variable to a given value as an atomic operation.

Syntax

```
LONG InterlockedExchange(
  LONG volatile *Target,
  LONG          Value
);
```

Parameters

`Target`

A pointer to a variable to be set to the supplied *Value* as an atomic operation.

`Value`

Specifies the value to which the variable will be set.

Return Value

InterlockedExchange returns the value of the variable at *Target* when the call occurred.

Remarks

InterlockedExchange should be used instead of **ExInterlockedExchangeUlong**, because it is both faster and more efficient.

InterlockedExchange is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

A call to **InterlockedExchange** routine is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedIncrement

InterlockedExchangeAdd function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchangeAdd** routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.

Syntax

```
LONG InterlockedExchangeAdd(
  LONG volatile *Addend,
  LONG          Value
);
```

Parameters

`Addend`

A pointer to an integer variable.

`Value`

Specifies the value to be added to *Addend*.

Return Value

InterlockedExchangeAdd returns the original value of the *Addend* variable when the call occurred.

Remarks

InterlockedExchangeAdd should be used instead of **ExInterlockedAddUlong** because it is both faster and more efficient.

InterlockedExchangeAdd is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedExchangeAdd is atomic only with respect to other **InterlockedXxx** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10

IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedIncrement

InterlockedExchangePointer function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedExchangePointer** routine performs an atomic operation that sets a pointer to a new value.

Syntax

```
PVOID InterlockedExchangePointer(
  PVOID volatile *Target,
  PVOID          Value
);
```

Parameters

`Target`

A pointer to a PVOID value. The routine sets (**Target*) to *Value*.

`Value`

Specifies the PVOID value to set (**Target*) to.

Return Value

InterlockedExchangePointer returns the original value of the pointer at **Target* (that is, the value of this pointer at entry to the routine).

Remarks

InterlockedExchangePointer provides a fast, atomic way to synchronize updating a pointer variable that is shared by multiple threads.

InterlockedExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedExchangePointer** is atomic only with respect to other **Interlocked*Xxx*** calls. It does not use a spin lock and can be safely used on pageable data.

The *Target* parameter should be aligned on either a 32-bit or 64-bit boundary, depending on the system type, for better performance.

A call to **InterlockedExchangePointer** is atomic only with respect to other **Interlocked*Xxx*** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedCompareExchange

InterlockedCompareExchangePointer

InterlockedExchange

InterlockedIncrement function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedIncrement** routine increments a caller-supplied variable as an atomic operation.

Syntax

```
LONG CDECL_NON_WVMPURE InterlockedIncrement(
  LONG volatile *Addend
);
```

Parameters

`Addend`

A pointer to a variable of type LONG.

Return Value

InterlockedIncrement returns the incremented value.

Remarks

InterlockedIncrement should be used instead of **ExInterlockedIncrementLong** because it is both more efficient and faster.

InterlockedIncrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedIncrement is atomic only with respect to other **InterlockedXxx** calls.

Interlocked operations cannot be used on non-cached memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	OneCoreUAP.lib on Windows 10
IRQL	Any level

See Also

ExInterlockedAddLargeInteger

ExInterlockedAddUlong

InterlockedDecrement

InterlockedExchange

InterlockedOr function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedOr** routine atomically computes a bitwise OR operation.

Syntax

```
LONG InterlockedOr(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be ORed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be ORed with the variable that is pointed to by *Destination*.

Return Value

InterlockedOr returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedOr atomically computes ***Destination*|=*Value***.

Interlocked operations cannot be used on non-cached memory.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedAnd

InterlockedXor

InterlockedXor function

1/25/2019 • 2 minutes to read • Edit Online

The **InterlockedOr** routine atomically computes a bitwise exclusive OR operation.

Syntax

```
LONG InterlockedXor(
  LONG volatile *Destination,
  LONG          Value
);
```

Parameters

`Destination`

A pointer to the variable to be exclusive ORed with *Value*. The result of the operation is stored in the variable.

`Value`

Specifies the value to be exclusive ORed with the variable that is pointed to by *Destination*.

Return Value

InterlockedXor returns the original value stored in the variable pointed to by *Destination*.

Remarks

InterlockedXor atomically computes ***Destination*^=*Value***.

Interlocked operations cannot be used on non-cached memory.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
IRQL	Any level

See Also

InterlockedAnd

InterlockedOr

IO_COMPLETION_ROUTINE callback function

1/25/2019 • 3 minutes to read • Edit Online

The *IoCompletion* routine completes the processing of I/O operations.

Syntax

```
IO_COMPLETION_ROUTINE IoCompletionRoutine;

NTSTATUS IoCompletionRoutine(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  PVOID Context
)
{...}
```

Parameters

`DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`Irp`

Caller-supplied pointer to an IRP structure that describes the I/O operation.

`Context`

Caller-supplied pointer to driver-specific context information, previously supplied when calling IoSetCompletionRoutine or IoSetCompletionRoutineEx. Context information must be stored in nonpaged memory, since an *IoCompletion* routine can be called at DISPATCH_LEVEL. For more information, see the following Remarks section.

Return Value

If the *IoCompletion* routine determines that additional processing is required for the IRP, it must return STATUS_MORE_PROCESSING_REQUIRED. For more information, see the following Remarks section. Otherwise, it should return STATUS_SUCCESS. (The I/O manager only checks for the presence or absence of STATUS_MORE_PROCESSING_REQUIRED.)

Remarks

A driver's *IoCompletion* routine executes in an arbitrary thread or DPC context, and at an IRQL that is less than or equal to DISPATCH_LEVEL. Because code written to execute at DISPATCH_LEVEL will also execute at lower levels, *IoCompletion* routines should be designed for execution at DISPATCH_LEVEL. However, because these routines are not guaranteed to run at DISPATCH_LEVEL, they must not call system routines that actually require execution at DISPATCH_LEVEL. (For more information about IRQLs, see Managing Hardware Priorities.)

To register an *IoCompletion* routine for a specific IRP, a driver must call IoSetCompletionRoutine or IoSetCompletionRoutineEx, which stores the *IoCompletion* routine's address in the next-lower driver's I/O stack location. (Thus, a lowest-level driver cannot register an *IoCompletion* routine.) A driver typically calls

IoSetCompletionRoutine or **IoSetCompletionRoutineEx** from one of its dispatch routines, each time an IRP is received. Most drivers, including all PnP drivers, can use **IoSetCompletionRoutine** to register their *IoCompletion* routine. Non-PnP drivers that may be unloaded before their *IoCompletion* routine executes should use **IoSetCompletionRoutineEx** instead.

When any driver completes an IRP, it calls IoCompleteRequest, which in turn calls the *IoCompletion* routine of each higher-level driver, from the next-highest to the highest, until all higher *IoCompletion* routines have been called or until one routine returns STATUS_MORE_PROCESSING_REQUIRED.

When you create the IRP, allocate a stack location for the current driver as well as any lower drivers. If you do not allocate sufficient stack locations, the *DeviceObject* pointer might be set to **NULL** when the completion routine is called. You can avoid allocating extra stack location for the current driver if you use the *Context* field to pass information to *IoCompletion* rather then relying on the *DeviceObject* parameter.

If an *IoCompletion* routine returns STATUS_MORE_PROCESSING_REQUIRED, the lower driver's call to **IoCompleteRequest** immediately returns. In this case, a higher-level driver will have to call **IoCompleteRequest** to complete the IRP.

For more information about implementing *IoCompletion* routines, see Completing IRPs.

Examples

To define an *IoCompletion* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *IoCompletion* callback routine that is named `MyIoCompletion`, use the IO_COMPLETION_ROUTINE type as shown in this code example:

```
IO_COMPLETION_ROUTINE MyIoCompletion;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyIoCompletion(
    PDEVICE_OBJECT  DeviceObject,
    PIRP  Irp,
    PVOID  Context
    )
  {
      // Function body
  }
```

The IO_COMPLETION_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_COMPLETION_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL (see Remarks section).

IO_CSQ_ACQUIRE_LOCK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqAcquireLock* routine is used by the system to acquire the lock for a driver-implemented, cancel-safe IRP queue.

Syntax

```
IO_CSQ_ACQUIRE_LOCK IoCsqAcquireLock;

void IoCsqAcquireLock(
  PIO_CSQ Csq,
  PKIRQL Irql
)
{...}
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irql`

Pointer to a variable that the *CsqAcquireLock* routine can use to store the current IRQL. The system passes the stored value CsqReleaseLock when it releases the lock.

Return Value

None

Remarks

The driver specifies the *CsqAcquireLock* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqAcquireLock* parameter of IoCsqInitialize or IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The system calls this routine to acquire a lock on the driver's IRP queue before attempting to insert or remove an IRP from the queue. The system calls the CsqReleaseLock routine to release the lock.

If the driver uses a spin lock to implement locking for the queue, it must store the current IRQL for when it releases the spin lock. The system passes a pointer to an IRQL variable that the driver can use to store the current IRQL. The system passes the stored value as the *Irql* parameter to *CsqReleaseLock* when it releases the lock. Otherwise the driver can ignore the *Irql* parameter. For information about spin locks, see Spin Locks.

Drivers can use any locking mechanism to lock the queue, such as mutexes. For more information about mutexes, see Mutex Objects.

Examples

To define a *CsqAcquireLock* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification

tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqAcquireLock* callback routine that is named `MyCsqAcquireLock`, use the IO_CSQ_ACQUIRE_LOCK type as shown in this code example:

```
IO_CSQ_ACQUIRE_LOCK MyCsqAcquireLock;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
 MyCsqAcquireLock(
    PIO_CSQ  Csq,
    PKIRQL  Irql
    )
  {
      // Function body
  }
```

The IO_CSQ_ACQUIRE_LOCK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_ACQUIRE_LOCK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	See Remarks section.

See Also

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_COMPLETE_CANCELED_IRP callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqCompleteCanceledIrp* routine is used by the system to signal to the driver that it can complete a canceled IRP.

Syntax

```
IO_CSQ_COMPLETE_CANCELED_IRP IoCsqCompleteCanceledIrp;

void IoCsqCompleteCanceledIrp(
  PIO_CSQ Csq,
  PIRP Irp
)
{...}
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irp`

Pointer to the IRP to be canceled.

Return Value

None

Remarks

The driver specifies the *CsqCompleteCanceledIrp* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqCompleteCanceledIrp* parameter of IoCsqInitialize or IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The system calls this routine to complete a canceled IRP that has been removed from the driver's queue. Normally, drivers just call IoCompleteRequest for the IRP with a status of STATUS_CANCELLED.

Drivers are not required to remove the IRP from the queue before completing it as canceled: the system always calls the queue's CsqRemoveIrp routine to remove the IRP from the queue before calling *CsqCompleteCanceledIrp*.

Examples

To define a *CsqCompleteCanceledIrp* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqCompleteCanceledIrp* callback routine that is named `MyCsqCompleteCanceledIrp`, use the IO_CSQ_COMPLETE_CANCELED_IRP type as shown in this code example:

```
IO_CSQ_COMPLETE_CANCELED_IRP MyCsqCompleteCanceledIrp;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
 MyCsqCompleteCanceledIrp(
    _In_ PIO_CSQ  Csq,
    _In_ PIRP  Irp
    )
  {
      // Function body
  }
```

The IO_CSQ_COMPLETE_CANCELED_IRP function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_COMPLETE_CANCELED_IRP function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCompleteRequest

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_INSERT_IRP callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqInsertIrp* routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.

Syntax

```
IO_CSQ_INSERT_IRP IoCsqInsertIrp;

void IoCsqInsertIrp(
  _IO_CSQ *Csq,
  PIRP Irp
)
{...}
```

Parameters

`*Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irp`

Pointer to the IRP to insert into the IRP queue.

Return Value

None

Remarks

The driver specifies the *CsqInsertIrp* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqInsertIrp* parameter of IoCsqInitialize when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The IoCsqInsertIrp and IoCsqInsertIrpEx routines call *CsqInsertIrp* to insert the IRP into the queue.

Examples

To define a *CsqInsertIrp* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqInsertIrp* callback routine that is named `MyCsqInsertIrp`, use the IO_CSQ_INSERT_IRP type as shown in this code example:

```
IO_CSQ_INSERT_IRP MyCsqInsertIrp;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
 MyCsqInsertIrp(
    struct _IO_CSQ  *Csq,
    PIRP  Irp
    )
  {
      // Function body
  }
```

The IO_CSQ_INSERT_IRP function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_INSERT_IRP function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_INSERT_IRP_EX callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqInsertIrpEx* routine is used by the system to insert an IRP into a driver-implemented, cancel-safe IRP queue.

Syntax

```
IO_CSQ_INSERT_IRP_EX IoCsqInsertIrpEx;

NTSTATUS IoCsqInsertIrpEx(
  _IO_CSQ *Csq,
  PIRP Irp,
  PVOID InsertContext
)
{...}
```

Parameters

`*Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irp`

Pointer to the IRP to insert into the IRP queue.

`InsertContext`

Pointer to driver-defined context for the insert operation.

Return Value

CsqInsertIrpEx returns STATUS_SUCCESS if the IRP has been successfully inserted, or the appropriate error code on failure.

Remarks

The driver specifies the *CsqInsertIrpEx* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqInsertIrp* parameter of IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The IoCsqInsertIrpEx routine calls *CsqInsertIrpEx* to insert the IRP into the queue. The *InsertContext* parameter of **IoCsqInsertIrpEx** is passed as the *InsertContext* parameter of *CsqInsertIrpEx*.

Examples

To define a *CsqInsertIrpEx* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqInsertIrpEx* callback routine that is named `MyCsqInsertIrpEx`, use the IO_CSQ_INSERT_IRP_EX type as shown in this code example:

```
IO_CSQ_INSERT_IRP_EX MyCsqInsertIrpEx;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
 MyCsqInsertIrpEx(
    struct _IO_CSQ  *Csq,
    PIRP  Irp,
    PVOID  InsertContext
    )
  {
      // Function body
  }
```

The IO_CSQ_INSERT_IRP_EX function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_INSERT_IRP_EX function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_PEEK_NEXT_IRP callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqPeekNextIrp* routine is used by the system to find the next matching IRP in a driver-implemented, cancel-safe IRP queue.

Syntax

```
IO_CSQ_PEEK_NEXT_IRP IoCsqPeekNextIrp;

PIRP IoCsqPeekNextIrp(
  PIO_CSQ Csq,
  PIRP Irp,
  PVOID PeekContext
)
{...}
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irp`

Pointer to an IRP in the driver's IRP queue, or **NULL** to specify the beginning of the queue.

`PeekContext`

Pointer to driver-defined context information. The *CsqPeekNextIrp* routine returns the first IRP that follows *Irp* and matches *PeekContext*.

Return Value

CsqPeekNextIrp returns the first IRP that follows *Irp* in the queue and matches *PeekContext*, or **NULL** if there is no matching IRP. If *Irp* is **NULL**, *CsqPeekNextIrp* returns the first matching IRP in the queue, or **NULL** if there is no matching IRP.

Remarks

The driver specifies the *CsqPeekNextIrp* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqPeekNextIrp* parameter of IoCsqInitialize or IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The system uses *CsqPeekNextIrp* to iterate through the IRPs in the driver's IRP queue. For example, the IoCsqRemoveNextIrp routine uses this routine to find the IRP to be removed.

Examples

To define a *CsqPeekNextIrp* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqPeekNextIrp* callback routine that is named `MyCsqPeekNextIrp`, use the IO_CSQ_PEEK_NEXT_IRP type as shown in this code example:

```
IO_CSQ_PEEK_NEXT_IRP MyCsqPeekNextIrp;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
PIRP
 MyCsqPeekNextIrp(
    PIO_CSQ  Csq,
    PIRP  Irp,
    PVOID  PeekContext
    )
  {
      // Function body
  }
```

The IO_CSQ_PEEK_NEXT_IRP function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_PEEK_NEXT_IRP function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_RELEASE_LOCK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqReleaseLock* routine is used by the system to release the lock that was acquired using CsqAcquireLock.

Syntax

```
IO_CSQ_RELEASE_LOCK IoCsqReleaseLock;

void IoCsqReleaseLock(
  PIO_CSQ Csq,
  KIRQL Irql
)
{...}
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irql`

Specifies an IRQL. This is the value stored by CsqAcquireLock when the lock was acquired.

Return Value

None

Remarks

The driver specifies the *CsqReleaseLock* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqReleaseLock* parameter of IoCsqInitialize or IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The system calls this function to release a lock that was acquired using CsqAcquireLock.

If the driver uses a spin lock to implement locking for the queue, it must store the current IRQL when it acquires the lock, and provide the stored IRQL when it releases the lock. The *CsqAcquireLock* routine stores the current IRQL, and the system passes the stored value as the *Irql* parameter to *CsqReleaseLock*. Otherwise the driver can ignore the *Irql* parameter. For information about spin locks, see Spin Locks.

Drivers can use any locking mechanism to lock the queue, such as mutexes. For more information about mutexes, see Mutex Objects.

Examples

To define a *CsqReleaseLock* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqReleaseLock* callback routine that is named `MyCsqReleaseLock`, use the IO_CSQ_RELEASE_LOCK type as shown in this code example:

```
IO_CSQ_RELEASE_LOCK MyCsqReleaseLock;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
 MyCsqReleaseLock(
    PIO_CSQ  Csq,
    KIRQL  Irql
    )
  {
      // Function body
  }
```

The IO_CSQ_RELEASE_LOCK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_RELEASE_LOCK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_CSQ_REMOVE_IRP callback function

1/25/2019 • 2 minutes to read • Edit Online

The *CsqRemoveIrp* routine is used by the system to remove the specified IRP from a driver-implemented, cancel-safe IRP queue.

Syntax

```
IO_CSQ_REMOVE_IRP IoCsqRemoveIrp;

void IoCsqRemoveIrp(
  PIO_CSQ Csq,
  PIRP Irp
)
{...}
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the cancel-safe IRP queue.

`Irp`

Pointer to the IRP to remove from the IRP queue.

Return Value

None

Remarks

The driver specifies the *CsqRemoveIrp* routine for a cancel-safe IRP queue when it initializes the queue's **IO_CSQ** structure. The driver specifies the routine as the *CsqRemoveIrp* parameter of IoCsqInitialize or IoCsqInitializeEx when it initializes **IO_CSQ**. For more information, see Cancel-Safe IRP Queues.

The IoCsqRemoveIrp and IoCsqRemoveNextIrp routines call the cancel-safe IRP queue's *CsqRemoveIrp* routine to remove the specified IRP from the queue. The system also uses *CsqRemoveIrp* to remove a canceled IRP from the queue.

Examples

To define a *CsqRemoveIrp* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CsqRemoveIrp* callback routine that is named `MyCsqRemoveIrp`, use the IO_CSQ_REMOVE_IRP type as shown in this code example:

```
```

```
IO_CSQ_REMOVE_IRP MyCsqRemoveIrp;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
 MyCsqRemoveIrp(
    PIO_CSQ  Csq,
    PIRP  Irp
    )
  {
      // Function body
  }
```

The IO_CSQ_REMOVE_IRP function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_CSQ_REMOVE_IRP function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IO_DPC_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DpcForIsr* routine finishes the servicing of an I/O operation, after an InterruptService routine returns.

Syntax

```
IO_DPC_ROUTINE IoDpcRoutine;

void IoDpcRoutine(
  PKDPC Dpc,
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp,
  PVOID Context
)
{...}
```

Parameters

`Dpc`

Caller-supplied pointer to a KDPC structure, which represents the DPC object that is associated with this *DpcForIsr* routine.

`*DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`*Irp`

Caller-supplied pointer to an IRP structure that describes the I/O operation.

`Context`

Caller-supplied pointer to driver-defined context information, specified in a previous call to IoRequestDpc.

Return Value

None

Remarks

To register a *DpcForIsr* routine for a specific device object, a driver must call IoInitializeDpcRequest, which causes the system to allocate and initialize one DPC object. (If you need multiple DPC routines, use CustomDpc routines.)

To queue a *DpcForIsr* routine for execution, a driver's InterruptService routine must call IoRequestDPC.

A *DpcForIsr* routine is typically responsible for at least the following tasks:

- Completing the I/O operation described by the received IRP.
- Dequeuing next IRP.

  If the driver uses the system-supplied IRP queue, the *DpcForIsr* routine should call IoStartNextPacket or

IoStartNextPacketByKey, so the driver's StartIo routine will start processing the next I/O request.

If the driver uses internal IRP queues, the *DpcForIsr* routine should dequeue the next IRP and begin processing for the next I/O request.

- Setting the I/O status block in the received IRP and calling IoCompleteRequest for the completed request.

A *DpcForIsr* routine might also retry a failed operation or set up the next transfer for a large I/O request that has been broken into smaller pieces.

For more information about *DpcForIsr* routines, see DPC Objects and DPCs.

Examples

To define a *DpcForIsr* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DpcForIsr* callback routine that is named `MyDpcForIsr`, use the IO_DPC_ROUTINE type as shown in this code example:

```
IO_DPC_ROUTINE MyDpcForIsr;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDpcForIsr(
    PKDPC  Dpc,
    struct _DEVICE_OBJECT  *DeviceObject,
    struct _IRP  *Irp,
    PVOID  Context
    )
  {
      // Function body
  }
```

The IO_DPC_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_DPC_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

IRQL	Called at DISPATCH_LEVEL.

IO_SESSION_NOTIFICATION_FUNCTION callback function

1/25/2019 • 2 minutes to read • Edit Online

The IO_SESSION_NOTIFICATION_FUNCTION function type defines a callback routine through which a driver receives notifications of changes in the state of user sessions that the driver is interested in.

Syntax

```
IO_SESSION_NOTIFICATION_FUNCTION IoSessionNotificationFunction;

NTSTATUS IoSessionNotificationFunction(
  PVOID SessionObject,
  PVOID IoObject,
  ULONG Event,
  PVOID Context,
  PVOID NotificationPayload,
  ULONG PayloadLength
)
{...}
```

Parameters

`SessionObject`

Pointer to an opaque, system object that contains information about the user session. The driver can pass this pointer value to the IoGetContainerInformation routine as the *ContainerObject* parameter value.

`IoObject`

Pointer to an I/O object owned by the driver. This parameter is the I/O object pointer that the driver supplied to the IoRegisterContainerNotification routine when the driver previously registered to receive notifications of session events. The **IoRegisterContainerNotification** routine's *NotificationInformation* parameter points to an IO_SESSION_STATE_NOTIFICATION structure whose **IoObject** member points to the I/O object.

`Event`

An IO_SESSION_EVENT enumeration constant that indicates which session event caused the notification callback.

`Context`

The context value that the driver previously supplied to the **IoRegisterContainerNotification** routine when the driver registered to receive notifications of session events. In the **IoRegisterContainerNotification** call, the driver supplied a pointer to an **IO_SESSION_STATE_NOTIFICATION** structure whose **Context** member contains the context value.

`NotificationPayload`

Pointer to a payload buffer that contains an IO_SESSION_CONNECT_INFO structure.

`PayloadLength`

The size, in bytes, of the buffer pointed to by *NotificationPayload*. The buffer size never needs exceeds the constant value IO_SESSION_MAX_PAYLOAD_SIZE, which is defined in the Wdm.h header file.

Return Value

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in the Ntstatus.h header file.

Remarks

A kernel-mode driver implements this routine. The I/O manager calls this routine to notify the driver of session events.

To receive notifications of session events, a driver calls the **IoRegisterContainerNotification** routine and sets this routine's *CallbackFunction* parameter to point to the driver's IO_SESSION_NOTIFICATION_FUNCTION routine. The I/O object that the driver passes to **IoRegisterContainerNotification** determines whether the driver will receive notifications of events in a particular user session or of events in all sessions. For more information, see IO_SESSION_STATE_NOTIFICATION.

Examples

To define an an I/O-session notification routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an an I/O-session notification routine that is named `MyIoSessionNotification`, use the IO_SESSION_NOTIFICATION_FUNCTION type as shown in this code example:

```
IO_SESSION_NOTIFICATION_FUNCTION MyIoSessionNotification;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyIoSessionNotification(
    PVOID  SessionObject,
    PVOID  IoObject,
    ULONG  Event,
    PVOID  Context,
    PVOID  NotificationPayload,
    ULONG  PayloadLength
    )
  {
      // Function body
  }
```

The IO_SESSION_NOTIFICATION_FUNCTION function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_SESSION_NOTIFICATION_FUNCTION function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows 7 and later versions of the Windows operating system.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
IRQL	Called at IRQL <= APC_LEVEL.

See Also

IO_SESSION_CONNECT_INFO

IO_SESSION_EVENT

IO_SESSION_STATE_NOTIFICATION

IoGetContainerInformation

IoRegisterContainerNotification

IO_TIMER_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *IoTimer* routine is a DPC that, if registered, is called once per second.

Syntax

```
IO_TIMER_ROUTINE IoTimerRoutine;

void IoTimerRoutine(
  _DEVICE_OBJECT *DeviceObject,
  PVOID Context
)
{...}
```

Parameters

`*DeviceObject`

Caller-supplied pointer to a DEVICE_OBJECT structure. This is the device object for the target device, previously created by the driver's AddDevice routine.

`Context`

Caller-supplied pointer to driver-defined context information, specified in a previous call to IoInitializeTimer.

Return Value

None

Remarks

A driver's *IoTimer* routine executes in a DPC context, at IRQL = DISPATCH_LEVEL.

A driver can associate an *IoTimer* routine with each device object it creates. (You can use a single *IoTimer* routine with multiple device objects, or a separate routine with each device object.) To register an *IoTimer* routine, a driver must call IoInitializeTimer, supplying the *IoTimer* routine's address and a device object pointer.

To queue an *IoTimer* routine for execution, a driver routine must call IoStartTimer. The system calls the *IoTimer* routine once per second until the driver calls IoStopTimer.

For more information about *IoTimer* routines, see IoTimer Routines.

Examples

To define an *IoTimer* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *IoTimer* callback routine that is named `MyIoTimer`, use the IO_TIMER_ROUTINE type as shown in this code example:

```
IO_TIMER_ROUTINE MyIoTimer;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyIoTimer(
    struct DEVICE_OBJECT  *DeviceObject,
    PVOID  Context
    )
  {
      // Function body
  }
```

The IO_TIMER_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_TIMER_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL (see Remarks section).

IO_WORKITEM_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

A *WorkItem* routine performs the processing for a work item that was queued by the IoQueueWorkItem routine.

Syntax

```
IO_WORKITEM_ROUTINE IoWorkitemRoutine;

void IoWorkitemRoutine(
  PDEVICE_OBJECT DeviceObject,
  PVOID Context
)
{...}
```

Parameters

`DeviceObject`

Pointer to one of the caller's device objects. This is the pointer that was passed as the *DeviceObject* parameter to IoAllocateWorkItem when the work item was allocated, or as the *IoObject* parameter to IoInitializeWorkItem when the work item was initialized.

`Context`

Specifies driver-specific context information. This is the value that was passed as the *Context* parameter to IoQueueWorkItem when the work item was queued.

Return Value

None

Remarks

The driver queues a *WorkItem* routine by calling IoQueueWorkItem, and a system worker thread subsequently executes the routine. A *WorkItem* routine must run for a limited amount of time; otherwise, the system can deadlock. For more information, see System Worker Threads.

A *WorkItem* routine runs at IRQL = PASSIVE_LEVEL and in a system thread context.

Examples

To define a *WorkItem* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *WorkItem* callback routine that is named `MyWorkItem`, use the IO_WORKITEM_ROUTINE type as shown in this code example:

```
```

```
IO_WORKITEM_ROUTINE MyWorkItem;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyWorkItem(
    PDEVICE_OBJECT  DeviceObject,
    PVOID  Context
    )
  {
      // Function body
  }
```

The IO_WORKITEM_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_WORKITEM_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IO_WORKITEM

IoQueueWorkItem

IO_WORKITEM_ROUTINE_EX callback function

1/25/2019 • 2 minutes to read • Edit Online

A *WorkItemEx* routine performs the processing for a work item that was queued by the IoQueueWorkItemEx or IoTryQueueWorkItem routine.

Syntax

```
IO_WORKITEM_ROUTINE_EX IoWorkitemRoutineEx;

void IoWorkitemRoutineEx(
  PVOID IoObject,
  PVOID Context,
  PIO_WORKITEM IoWorkItem
)
{...}
```

Parameters

`IoObject`

Pointer to the caller's driver object or to one of the caller's device objects. This is the pointer that was passed as the *DeviceObject* parameter to IoAllocateWorkItem when the work item was allocated, or as the *IoObject* parameter to IoInitializeWorkItem or IoTryQueueWorkItem when the work item was initialized.

`Context`

Specifies driver-specific context information. This is the value that was passed as the *Context* parameter to IoQueueWorkItemEx or IoTryQueueWorkItem when the work item was queued.

`IoWorkItem`

Pointer to the IO_WORKITEM structure for the work item. This is the pointer that was passed as the *IoWorkItem* parameter to IoQueueWorkItemEx or IoTryQueueWorkItem.

Return Value

None

Remarks

Drivers can implement *WorkItemEx* routines only on Windows Vista and later versions of Windows.

The driver queues a *WorkItemEx* routine by calling IoQueueWorkItemEx or IoTryQueueWorkItem, and a system worker thread subsequently executes the routine. For more information, see System Worker Threads.

A *WorkItemEx* routine must run for a limited amount of time; otherwise, the system can deadlock. For more information, see System Worker Threads.

A *WorkItemEx* routine runs at IRQL = PASSIVE_LEVEL and in a system thread context.

Examples

To define a *WorkItemEx* callback routine, you must first provide a function declaration that identifies the type of

callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *WorkItemEx* callback routine that is named `MyWorkItemEx`, use the IO_WORKITEM_ROUTINE_EX type as shown in this code example:

```
IO_WORKITEM_ROUTINE_EX MyWorkItemEx;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyWorkItemEx(
    PVOID  IoObject,
    PVOID  Context,
    PIO_WORKITEM  IoWorkItem
    )
  {
      // Function body
  }
```

The IO_WORKITEM_ROUTINE_EX function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the IO_WORKITEM_ROUTINE_EX function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IO_WORKITEM

IoInitializeWorkItem

IoQueueWorkItemEx

IoTryQueueWorkItem

IoAcquireRemoveLock macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoAcquireRemoveLock** routine increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack or deleted.

Syntax

```
void IoAcquireRemoveLock(
   RemoveLock,
   Tag
);
```

Parameters

`RemoveLock`

Pointer to an **IO_REMOVE_LOCK** structure that the caller initialized with a previous call to IoInitializeRemoveLock.

`Tag`

Optionally points to a caller-supplied tag that identifies this instance of acquiring the remove lock. For example, a driver Dispatch routine typically sets this parameter to a pointer to the IRP the routine is processing.

If a driver specifies a *Tag* on a call to **IoAcquireRemoveLock**, the driver must supply the same *Tag* in the corresponding call to IoReleaseRemoveLock.

The *Tag* does not have to be unique, but should be something meaningful during debugging.

The I/O system uses this parameter on checked builds only.

Return Value

This macro wraps and assumes the return value of **IoAcquireRemoveLockEx**, which is NTSTATUS.

IoAcquireRemoveLock returns STATUS_SUCCESS if the call was successful. One possible error return value is the following:

STATUS_DELETE_PENDING: The driver has received an IRP_MN_REMOVE_DEVICE for the device and has called IoReleaseRemoveLockandWait. That routine is waiting for all remove locks to clear before returning control to the driver.

If the routine returns any value besides STATUS_SUCCESS, do not start any new operations on the device.

Remarks

A driver must initialize a remove lock with a call to IoInitializeRemoveLock before using the lock.

A driver must call IoReleaseRemoveLock to release the lock when it is no longer needed.

For more information, see Using Remove Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	CompleteRequestStatusCheck, MarkDevicePower, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, MultRemoveLock, NsRemoveLockMnRemove, NsRemoveLockMnSurpriseRemove, NsRemoveLockQueryMnRemove, PowerDownAllocate, PowerDownFail, PowerUpFail, RemoveLock, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove, RemoveLockMnRemove2, RemoveLockMnSurpriseRemove, RemoveLockQueryMnRemove, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePnp, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite, WmiForward

See Also

IoInitializeRemoveLock

IoReleaseRemoveLock

IoReleaseRemoveLockAndWait

IoAdjustPagingPathCount macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoAdjustPagingPathCount** routine increments or decrements a caller-supplied page-file counter as an atomic operation.

Syntax

```
void IoAdjustPagingPathCount(
   _count_,
   _paging_
);
```

Parameters

`_count_`

`_paging_`

[in]

Specifies whether the counter is to be incremented or decremented. A value of **TRUE** specifies an increment operation.

Return Value

None

Remarks

This routine is useful for maintaining a count of paging files on a device. The operating system notifies a driver that a paging file has been created on, or removed from, one of the driver's devices by sending an IRP. The IRP has the major code IRP_MJ_PNP and the minor code IRP_MN_DEVICE_USAGE_NOTIFICATION.

This routine can be used for other counters, such as counters for hibernation files or crash-dump files.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IRP_MJ_PNP

IRP_MN_DEVICE_USAGE_NOTIFICATION

IoAllocateDriverObjectExtension function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateDriverObjectExtension** routine allocates a per-driver context area, called a driver object extension, and assigns a unique identifier to it.

Syntax

```
NTSTATUS IoAllocateDriverObjectExtension(
  PDRIVER_OBJECT DriverObject,
  PVOID          ClientIdentificationAddress,
  ULONG          DriverObjectExtensionSize,
  PVOID          *DriverObjectExtension
);
```

Parameters

`DriverObject`

Pointer to a driver object to which the context area will be associated.

`ClientIdentificationAddress`

Specifies a unique identifier for the context area to be allocated.

`DriverObjectExtensionSize`

Specifies the length, in bytes, of the context area to be allocated.

`DriverObjectExtension`

Pointer to, on completion, the allocated context area.

Return Value

IoAllocateDriverObjectExtension returns one of the following NTSTATUS codes:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	Indicates that the routine allocated an extension of the requested size.

STA TUS _IN SUF FICI ENT _RE SO URC ES	Indicates that the memory could not be allocated for the driver object extension.
STA TUS _OB JEC T_N AM E_C OLL ISIO N	Indicates that a driver object extension with the given *ClientIdentificationAddress* already exists.

Remarks

Memory allocated by the system for the driver object extension is resident storage and is accessible from any IRQL. The allocated storage is automatically freed by the I/O manager when the driver object is deleted.

Callers of this routine must provide a unique identifier for *ClientIdentificationAddress*. To retrieve a pointer to the context area, a caller passes the *ClientIdentificationAddress* to IoGetDriverObjectExtension.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoGetDriverObjectExtension

IoAllocateErrorLogEntry function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateErrorLogEntry** routine allocates an error log entry, and returns a pointer to the packet that the caller uses to supply information about an I/O error.

Syntax

```
PVOID IoAllocateErrorLogEntry(
  PVOID IoObject,
  UCHAR EntrySize
);
```

Parameters

`IoObject`

Pointer to a device object representing the device on which an I/O error occurred, or to a driver object representing the driver that found an error.

`EntrySize`

Specifies the size, in bytes, of the error log entry to be allocated. This value cannot exceed ERROR_LOG_MAXIMUM_SIZE.

> **Warning** *EntrySize* is a UCHAR value. If you specify a larger value, the compiler will silently truncate that value to a (wrong) UCHAR.

Return Value

IoAllocateErrorLogEntry returns a pointer to the error log entry, or **NULL** if a log entry could not be allocated.

Remarks

The driver must first fill in the packet with information about the error, then call IoWriteErrorLogEntry to post the entry to the error log. The error log entry buffer is automatically freed once the log entry is recorded. Entry buffers that are not going to be written to the log can be freed by using IoFreeErrorLogEntry.

An error log entry consists of a variable-length IO_ERROR_LOG_PACKET structure, possibly followed by one or more zero-counted Unicode strings. The Event Viewer inserts these strings into the error message it displays for the entry. **IO_ERROR_LOG_PACKET** contains one variable-length member, the **DumpData** member. Thus, the value for *EntrySize* must be **sizeof**(**IO_ERROR_LOG_PACKET**) + size of the **DumpData** member + combined size of any driver-supplied insertion strings.

Drivers should check that the value for *EntrySize* is less than ERROR_LOG_MAXIMUM_SIZE before calling **IoAllocateErrorLogEntry**. Since *EntrySize* is declared as a UCHAR, and the compiler will silently truncate any value too big to fit into a UCHAR, the routine itself cannot reliably detect if the passed value is too large.

Drivers must not treat **IoAllocateErrorLogEntry** returning **NULL** as a fatal error. The driver must continue to function normally, regardless of whether it can log errors.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_ERROR_LOG_PACKET

IoFreeErrorLogEntry

IoWriteErrorLogEntry

IoAllocateIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateIrp** routine allocates an IRP, given the number of I/O stack locations for each driver layered under the caller, and, optionally, for the caller.

Syntax

```
PIRP IoAllocateIrp(
  CCHAR   StackSize,
  BOOLEAN ChargeQuota
);
```

Parameters

`StackSize`

Specifies the number of I/O stack locations to be allocated for the IRP. This value must be at least equal to the **StackSize** of the next-lower driver's device object, but can be one greater than this value. The calling driver need not allocate a stack location in the IRP for itself.

`ChargeQuota`

Setting this to **TRUE** causes the memory allocated for the IRP to be charged against the quota for the current process. Should be set to **FALSE** by intermediate drivers. This can be set to **TRUE** only by highest-level drivers that are called in the context of the thread that originates the I/O request for which the driver is allocating another IRP.

Return Value

IoAllocateIrp returns a pointer to an IRP, which was allocated from nonpaged system space, or **NULL** if an IRP could not be allocated.

Remarks

The **IoAllocateIrp** routine does not associate the IRP with a thread. The allocating driver must free the IRP instead of completing it back to the I/O manager.

An intermediate or highest-level driver can call **IoAllocateIrp** to create IRPs for requests it sends to lower-level drivers. Such a driver must initialize the IRP and must set its IoCompletion routine in the IRP it creates so the caller can dispose of the IRP when lower-level drivers have completed processing of the request.

IoAllocateIrp automatically initializes the IRP's members. Do not use **IoInitializeIrp** to initialize the IRP before its first use. (You can use **IoInitializeIrp** to reuse an IRP that you have already used under certain special circumstances. See Reusing IRPs for details.)

An intermediate or highest-level driver also can call IoBuildDeviceIoControlRequest, IoBuildAsynchronousFsdRequest or IoBuildSynchronousFsdRequest to set up requests it sends to lower-level drivers. Only a highest-level driver can call IoMakeAssociatedIrp.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	ForwardedAtBadIrqlAllocate, IoAllocateComplete, IoAllocateForward, IoAllocateFree, IoAllocateIrpSignalEventInCompletion, IoAllocateIrpSignalEventInCompletion2, IoAllocateIrpSignalEventInCompletion3, IoAllocateIrpSignalEventInCompletionTimeout, IoReuseIrp, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, HwStorPortProhibitedDDIs, IoFreeIrp, SpNoWait, StorPortStartIo

See Also

IO_STACK_LOCATION

IRP

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoFreeIrp

IoMakeAssociatedIrp

IoReuseIrp

IoSetCompletionRoutine

IoAllocateMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateMdl** routine allocates a memory descriptor list (MDL) large enough to map a buffer, given the buffer's starting address and length. Optionally, this routine associates the MDL with an IRP.

Syntax

```
PMDL IoAllocateMdl(
  __drv_aliasesMem PVOID VirtualAddress,
  ULONG                  Length,
  BOOLEAN                SecondaryBuffer,
  BOOLEAN                ChargeQuota,
  PIRP                   Irp
);
```

Parameters

`VirtualAddress`

Pointer to the base virtual address of the buffer the MDL is to describe.

`Length`

Specifies the length, in bytes, of the buffer that the MDL is to describe. For more information, see the following Remarks section.

`SecondaryBuffer`

Indicates whether the buffer is a primary or secondary buffer. This parameter determines how the MDL is to be linked to the IRP. All buffers except the first buffer described by an MDL in an IRP are considered secondary buffers. This field must be **FALSE** if no IRP is associated with the MDL. For more information, see the following Remarks section.

`ChargeQuota`

Reserved for system use. Drivers must set this parameter to **FALSE**.

`Irp`

Pointer to an IRP to be associated with the MDL. If the *Irp* pointer is non-**NULL**, the allocated MDL is associated with the specified IRP's MDL list, according to the value of *SecondaryBuffer*.

Return Value

IoAllocateMdl returns a pointer to an MDL, or, if the MDL cannot be allocated, it returns **NULL**.

Remarks

IoAllocateMdl can be used by a driver that needs to break a buffer into pieces, each mapped by a separate MDL, or to map a driver-allocated buffer. The driver should call MmBuildMdlForNonPagedPool with the MDL allocated by this call to set up an MDL describing a driver-allocated buffer in nonpaged pool.

The *Length* parameter specifies the size of the buffer that is to be described by the MDL. In Windows Server 2003, Windows XP, and Windows 2000, the maximum buffer size, in bytes, that this routine can allocate is PAGE_SIZE * (65535 - **sizeof**(MDL)) / **sizeof**(ULONG_PTR). In Windows Vista and Windows Server 2008, the maximum buffer size is (2 gigabytes - PAGE_SIZE). In Windows 7 and Windows Server 2008 R2, the maximum buffer size is (4 gigabytes - PAGE_SIZE).

If the *SecondaryBuffer* parameter is **FALSE**, the routine updates *Irp*->**MdlAddress** to point to the new MDL. If *SecondaryBuffer* is **TRUE**, the routine adds the MDL to the end of the MDL chain that *Irp*->**MdlAddress** points to.

For more information about MDLs, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoBuildPartialMdl

IoFreeMdl

MmBuildMdlForNonPagedPool

IoAllocateWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAllocateWorkItem** routine allocates a work item.

Syntax

```
PIO_WORKITEM IoAllocateWorkItem(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the caller's driver object or to one of the caller's device objects. If the caller will later pass the work item to IoQueueWorkItem, *DeviceObject* must point to a device object.

Return Value

IoAllocateWorkItem returns a pointer to the allocated IO_WORKITEM structure. The routine returns **NULL** if sufficient resources do not exist.

Remarks

The driver must free the work item that is returned by **IoAllocateWorkItem** by calling IoFreeWorkItem.

IoAllocateWorkItem both allocates and initializes a work item. A related routine, IoInitializeWorkItem, initializes a work item in storage that the driver has previously allocated. Do not call **IoInitializeWorkItem** to initialize a work item that was allocated by **IoAllocateWorkItem**.

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of the Windows operating system, and in Windows Me. This routine is not available in Windows 98.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_WORKITEM

IoFreeWorkItem

IoInitializeWorkItem

IoQueueWorkItem

IoQueueWorkItemEx

IoAttachDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAttachDevice** routine attaches the caller's device object to a named target device object, so that I/O requests bound for the target device are routed first to the caller.

Syntax

```
NTSTATUS IoAttachDevice(
  PDEVICE_OBJECT  SourceDevice,
  PUNICODE_STRING TargetDevice,
  PDEVICE_OBJECT  *AttachedDevice
);
```

Parameters

`SourceDevice`

Pointer to the caller-created device object.

`TargetDevice`

Pointer to a buffer containing the name of the device object to which the specified *SourceDevice* is to be attached.

`AttachedDevice`

Pointer to caller-allocated storage for a pointer. On return, contains a pointer to the target device object if the attachment succeeds.

Return Value

IoAttachDevice can return one of the following NTSTATUS values:

Remarks

IoAttachDevice establishes layering between drivers so that the same IRPs can be sent to each driver in the chain.

This routine is used by intermediate drivers during initialization. It allows such a driver to attach its own device object to another device in such a way that any requests being made to the original device are given first to the intermediate driver.

The caller can be layered only at the top of an existing chain of layered drivers. **IoAttachDevice** searches for the highest device object layered over *TargetDevice* and attaches to that object (that can be the *TargetDevice*). Therefore, this routine must not be called if a driver that must be higher-level has already layered itself over the target device.

Note that for file system drivers and drivers in the storage stack, **IoAttachDevice** opens the target device with FILE_READ_ATTRIBUTES and then calls IoGetRelatedDeviceObject. This does not cause a file system to be mounted. Thus, a successful call to **IoAttachDevice** returns the device object of the storage driver, not that of the file system driver.

This routine sets the **AlignmentRequirement** in *SourceDevice* to the value in the next-lower device object and sets the **StackSize** to the value in the next-lower object plus one.

Warning *AttachedDevice* must point to a global memory location, such as the driver's device extension. **IoAttachDevice** opens the file object for the target device, updates *AttachedDevice*, performs the attach, and then closes the file object. Thus, the source device receives the IRP_MJ_CLEANUP and IRP_MJ_CLOSE requests for the file object before **IoAttachDevice** returns. The driver must forward these requests to the target device, and *AttachedDevice* must be a memory location accessible to the driver's DispatchCleanup and DispatchClose routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive1, HwStorPortProhibitedDDIs

See Also

DEVICE_OBJECT

IoAttachDeviceToDeviceStack

IoAttachDeviceToDeviceStackSafe

IoCreateDevice

IoDetachDevice

IoGetRelatedDeviceObject

IoAttachDeviceToDeviceStack function

1/25/2019 • 2 minutes to read • Edit Online

The **IoAttachDeviceToDeviceStack** routine attaches the caller's device object to the highest device object in the chain and returns a pointer to the previously highest device object.

Syntax

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack(
  PDEVICE_OBJECT SourceDevice,
  PDEVICE_OBJECT TargetDevice
);
```

Parameters

`SourceDevice`

Pointer to the caller-created device object.

`TargetDevice`

Pointer to another driver's device object, such as a pointer returned by a preceding call to IoGetDeviceObjectPointer.

Return Value

IoAttachDeviceToDeviceStack returns a pointer to the device object to which the *SourceDevice* was attached. The returned device object pointer can differ from *TargetDevice* if *TargetDevice* had additional drivers layered on top of it.

IoAttachDeviceToDeviceStack returns **NULL** if it could not attach the device object because, for example, the target device was being unloaded.

Remarks

IoAttachDeviceToDeviceStack establishes layering between drivers so that the same IRPs are sent to each driver in the chain.

An intermediate driver can use this routine during initialization to attach its own device object to another driver's device object. Subsequent I/O requests sent to *TargetDevice* are sent first to the intermediate driver.

This routine sets the **AlignmentRequirement** in *SourceDevice* to the value in the next-lower device object and sets the **StackSize** to the value in the next-lower-object plus one.

A driver writer must take care to call this routine before any drivers that must layer on top of their driver. **IoAttachDeviceToDeviceStack** attaches *SourceDevice* to the highest device object currently layered in the chain and has no way to determine whether drivers are being layered in the correct order.

A driver that acquired a pointer to the target device by calling IoGetDeviceObjectPointer should call ObDereferenceObject with the file object pointer that was returned by **IoGetDeviceObjectPointer** to release its reference to the file object before it detaches its own device object, for example, when such a higher-level driver is unloaded.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	AddDevice, PnpSameDeviceObject, HwStorPortProhibitedDDIs

See Also

IoAttachDevice

IoDetachDevice

IoGetDeviceObjectPointer

ObDereferenceObject

IoBuildAsynchronousFsdRequest function

1/25/2019 • 4 minutes to read • Edit Online

The **IoBuildAsynchronousFsdRequest** routine allocates and sets up an IRP to be sent to lower-level drivers.

Syntax

```
__drv_aliasesMem PIRP IoBuildAsynchronousFsdRequest(
  ULONG            MajorFunction,
  PDEVICE_OBJECT   DeviceObject,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   StartingOffset,
  PIO_STATUS_BLOCK IoStatusBlock
);
```

Parameters

`MajorFunction`

The major function code to be set in the IRP. This code can be IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FLUSH_BUFFERS, or IRP_MJ_SHUTDOWN.

`DeviceObject`

A pointer to the next-lower driver's device object. This object represents the target device for the read, write, flush, or shutdown operation.

`Buffer`

A pointer to a buffer into which data is read or from which data is written. The value of this argument is **NULL** for flush and shutdown requests.

`Length`

The length, in bytes, of the buffer pointed to by *Buffer*. For devices such as disks, this value must be an integer multiple of the sector size. Starting with Windows 8, the sector size can be 4,096 or 512 bytes. In earlier versions of Windows, the sector size is always 512 bytes. This parameter is required for read and write requests, but must be zero for flush and shutdown requests.

`StartingOffset`

A pointer to the starting offset on the input/output media. The value of this argument is zero for flush and shutdown requests.

`IoStatusBlock`

A pointer to the address of an I/O status block in which the to-be-called drivers return final status about the requested operation.

Return Value

IoBuildAsynchronousFsdRequest returns a pointer to an IRP, or a **NULL** pointer if the IRP cannot be allocated.

Remarks

Intermediate or highest-level drivers can call **IoBuildAsynchronousFsdRequest** to set up IRPs for requests sent to lower-level drivers. The calling driver must supply an IoCompletion routine for the IRP, so the IRP can be deallocated with IoFreeIrp. For more information about IRP deallocation, see Examples.

The IRP that gets built contains only enough information to get the operation started and to complete the IRP. No other context information is tracked because an asynchronous request is context-independent.

Lower-level drivers might impose restrictions on parameters supplied to this routine. For example, disk drivers might require that values supplied for *Length* and *StartingOffset* be integer multiples of the device's sector size.

An intermediate or highest-level driver also can call IoBuildDeviceIoControlRequest, IoAllocateIrp, or IoBuildSynchronousFsdRequest to set up requests it sends to lower-level drivers. Only a highest-level driver can call IoMakeAssociatedIrp.

During an **IoBuildAsynchronousFsdRequest** call, the I/O manager sets the **Tail.Overlay.Thread** member of the IRP structure to point to the caller's thread object, but does not take a counted reference to the thread object on behalf of the caller. After the caller sends the IRP to the driver for the target device, this driver might use the **Tail.Overlay.Thread** member to access the thread object. For example, a storage driver might call the IoSetHardErrorOrVerifyDevice routine and supply a pointer to the IRP as an input parameter. During this call, **IoSetHardErrorOrVerifyDevice** uses the **Tail.Overlay.Thread** member to access the thread object. When the thread object is accessed in this way, the driver that called **IoBuildAsynchronousFsdRequest** to allocate the IRP is responsible for ensuring that the thread object stays valid while the IRP is being handled.

To keep the thread object valid, the driver that calls **IoBuildAsynchronousFsdRequest** can take a counted reference on the thread object before sending the IRP. For example, this driver can call the ObReferenceObjectByPointerWithTag routine and supply, as the *Object* parameter, the object pointer from the **Tail.Overlay.Thread** member of the **IRP** structure. Later, this driver's completion routine can dereference the object by calling a routine such as ObDereferenceObjectWithTag.

A driver might call **IoBuildAsynchronousFsdRequest** in one thread, and send the IRP allocated by this call in another thread. Before sending the IRP, this driver should set the **Tail.Overlay.Thread** member of the IRP to point to the thread object for the sending thread. Typically, the driver calls the PsGetCurrentThread routine to get the thread object pointer.

A driver that calls **IoBuildAsynchronousFsdRequest** to allocate an IRP does not necessarily need to take a counted reference on the thread object pointed to by the **Tail.Overlay.Thread** member of the IRP. The driver might instead use another technique to guarantee that this thread object remains valid while the IRP is being handled. For example, if the driver created the thread, the thread can wait until the IRP is completed to terminate itself.

Examples

Before calling IoFreeIrp, an additional step is required to free the buffer for an IRP built by **IoBuildAsynchronousFsdRequest** if the following are all true:

- The buffer was allocated from system memory pool.
- In the device object for the target device, the DO_DIRECT_IO flag is set in the **DeviceObject->Flags** field.
- The **Irp->MdlAddress** field is non-NULL.

Before freeing the buffer for this IRP, call the MmUnlockPages routine with **Irp->MdlAddress** as the parameter value. This call decrements the extra reference count that **IoBuildAsynchronousFsdRequest** added to the pool pages in the MDL. Otherwise, the subsequent call to IoFreeMdl will bug check because the reference count for these pool pages will be 2, not 1. The following code example shows the **MmUnlockPages**, **IoFreeMdl**, and **IoFreeIrp** calls for this case:

```
if (((DeviceObject->Flags & DO_DIRECT_IO) == DO_DIRECT_IO) &&
    (Irp->MdlAddress != NULL))
{
    MmUnlockPages(Irp->MdlAddress);
}

IoFreeMdl(Irp->MdlAddress);
IoFreeIrp(Irp);
```

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	ForwardedAtBadIrqlFsdAsync, IoBuildFsdComplete, IoBuildFsdForward, IoBuildFsdFree, HwStorPortProhibitedDDIs

See Also

IO_STACK_LOCATION

IRP

IoAllocateIrp

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoCallDriver

IoFreeIrp

IoFreeMdl

IoMakeAssociatedIrp

IoSetCompletionRoutine

IoSetHardErrorOrVerifyDevice

MmUnlockPages

ObDereferenceObjectWithTag

ObReferenceObjectByPointerWithTag

PsGetCurrentThread

IoBuildDeviceIoControlRequest function

1/25/2019 • 3 minutes to read • Edit Online

The **IoBuildDeviceIoControlRequest** routine allocates and sets up an IRP for a synchronously processed device control request.

Syntax

```
__drv_aliasesMem PIRP IoBuildDeviceIoControlRequest(
  ULONG            IoControlCode,
  PDEVICE_OBJECT   DeviceObject,
  PVOID            InputBuffer,
  ULONG            InputBufferLength,
  PVOID            OutputBuffer,
  ULONG            OutputBufferLength,
  BOOLEAN          InternalDeviceIoControl,
  PKEVENT          Event,
  PIO_STATUS_BLOCK IoStatusBlock
);
```

Parameters

`IoControlCode`

Supplies the I/O control code (IOCTL) to be used in the request. For information about device type-specific I/O control codes, see device type-specific sections in the Windows Driver Kit (WDK).

`DeviceObject`

Supplies a pointer to the DEVICE_OBJECT structure for the next-lower driver's device object, which represents the target device.

`InputBuffer`

Supplies a pointer to an input buffer to be passed to the lower driver, or **NULL** if the request does not pass input data to lower drivers.

`InputBufferLength`

Supplies the length, in bytes, of the input buffer. If *InputBuffer* is **NULL**, *InputBufferLength* must be zero.

`OutputBuffer`

Supplies a pointer to an output buffer in which the lower driver is to return data, or **NULL** if the request does not require lower drivers to return data.

`OutputBufferLength`

Supplies the length, in bytes, of the output buffer. If *OutputBuffer* is **NULL**, *OutputBufferLength* must be zero.

`InternalDeviceIoControl`

If **TRUE**, the routine sets the IRP's major function code to IRP_MJ_INTERNAL_DEVICE_CONTROL. Otherwise, the routine sets the IRP's major function code to IRP_MJ_DEVICE_CONTROL.

`Event`

Supplies a pointer to a caller-allocated and initialized event object. The I/O manager sets the event to the Signaled state when a lower-level driver completes the requested operation. After calling IoCallDriver, the driver can wait for the event object. The *Event* parameter is optional and can be set to NULL. However, if *Event* is NULL, the caller must supply an IoCompletion routine for the IRP to notify the caller when the operation completes.

`IoStatusBlock`

Specifies an I/O status block to be set when the request is completed by lower drivers.

Return Value

If the operation succeeds, **IoBuildDeviceIoControlRequest** returns a pointer to an IRP, with the next-lower driver's I/O stack location set up from the supplied parameters. Otherwise, the routine returns **NULL**.

Remarks

A driver can call **IoBuildDeviceIoControlRequest** to set up IRPs for device control requests that it synchronously sends to lower-level drivers.

After calling **IoBuildDeviceIoControlRequest** to create a request, the driver must call IoCallDriver to send the request to the next-lower driver. If **IoCallDriver** returns STATUS_PENDING, the driver must wait for the completion of the IRP by calling KeWaitForSingleObject on the given *Event*. Most drivers do not need to set an IoCompletion routine for the IRP.

IRPs that are created by **IoBuildDeviceIoControlRequest** must be completed by a driver's call to IoCompleteRequest. A driver that calls **IoBuildDeviceIoControlRequest** must not call IoFreeIrp, because the I/O manager frees these synchronous IRPs after **IoCompleteRequest** has been called.

IoBuildDeviceIoControlRequest queues the IRPs that it creates to an IRP queue that is specific to the current thread. If the thread exits, the I/O manager cancels the IRP.

If the caller supplies an *InputBuffer* or *OutputBuffer* parameter, this parameter must point to a buffer that resides in system memory. The caller is responsible for validating any parameter values that it copies into the input buffer from a user-mode buffer. The input buffer might contain parameter values that are interpreted differently depending on whether the originator of the request is a user-mode application or a kernel-mode driver. In the IRP that **IoBuildDeviceIoControlRequest** returns, the **RequestorMode** field is always set to **KernelMode**. This value indicates that the request, and any information contained in the request, is from a trusted, kernel-mode component.

If the caller cannot validate parameter values that it copies from a user-mode buffer to the input buffer, or if these values must not be interpreted as coming from a kernel-mode component, the caller should set the **RequestorMode** field in the IRP to **UserMode**. This setting informs the driver that handles the I/O control request that the buffer contains untrusted, user-mode data.

The actual method by which the contents of the *InputBuffer* and *OutputBuffer* parameters are stored in the IRP depends on the *TransferType* value for the IOCTL. For more information about this value, see Buffer Descriptions for I/O Control Codes.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IoAllocateIrpSignalEventInCompletion, IoAllocateIrpSignalEventInCompletion2, IoAllocateIrpSignalEventInCompletion3, IoBuildDeviceControlNoFree, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildDeviceIoControlSetEvent, IrqlIoPassive1, PowerIrpDDis, SignalEventInCompletion, HwStorPortProhibitedDDIs

See Also

IO_STACK_LOCATION

IRP

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildSynchronousFsdRequest

IoCallDriver

IoCompleteRequest

KeInitializeEvent

KeWaitForSingleObject

IoBuildPartialMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **IoBuildPartialMdl** routine builds a new memory descriptor list (MDL) that represents part of a buffer that is described by an existing MDL.

Syntax

```
void IoBuildPartialMdl(
  PMDL  SourceMdl,
  PMDL  TargetMdl,
  PVOID VirtualAddress,
  ULONG Length
);
```

Parameters

`SourceMdl`

A pointer to an MDL that describes the original buffer, of which a subrange is to be mapped.

`TargetMdl`

A pointer to a caller-allocated MDL. This MDL must be large enough to describe the pages in the subrange that are specified by *VirtualAddress* and *Length*.

`VirtualAddress`

A pointer to the base virtual address for the subrange to be described by the *TargetMdl*.

`Length`

Specifies the length, in bytes, to be mapped by the *TargetMdl*. This value, in combination with *VirtualAddress*, must specify a buffer that is a proper subrange of the buffer that is described by *SourceMdl*. If *Length* is zero, the subrange to be mapped starts at *VirtualAddress* and includes the remaining range described by the *SourceMdl*.

Return Value

None

Remarks

This routine builds a target MDL that describes a subrange of the buffer that is described by the source MDL. This subrange is specified by the *VirtualAddress* and *Length* parameters. The *SourceMdl* and *TargetMdl* parameters point to the source MDL and target MDL.

A driver can use **IoBuildPartialMdl** to split a large transfer request into smaller transfer requests. The physical pages that the source MDL describes must be locked before the driver calls **IoBuildPartialMdl**. Typically, the source MDL describes a buffer in user address space, and the driver calls the MmProbeAndLockPages routine to lock the pages in this buffer. However, the driver can build the source MDL from nonpaged memory by calling the MmBuildMdlForNonPagedPool, MmAllocatePagesForMdlEx, or MmAllocatePagesForMdl routine.

A driver can map a partial MDL into the system address space by using the MmGetSystemAddressForMdlSafe

macro. If a partial MDL is built from a source MDL that is already mapped into the system address space, **MmGetSystemAddressForMdlSafe** uses the existing source mapping. Otherwise, **MmGetSystemAddressForMdlSafe** creates a new mapping. To prevent this new mapping from being leaked, drivers must call MmPrepareMdlForReuse before reusing a partial MDL. In addition, the IoFreeMdl routine releases the system-address-space mapping for a partial MDL, if such a mapping exists.

For more information about MDLs, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"
DDI compliance rules	MdlAfterReqCompletedIntIoctlA, MdlAfterReqCompletedIoctlA, MdlAfterReqCompletedReadA, MdlAfterReqCompletedWriteA

See Also

IoFreeMdl

MmAllocatePagesForMdl

MmAllocatePagesForMdlEx

MmBuildMdlForNonPagedPool

MmGetSystemAddressForMdlSafe

MmPrepareMdlForReuse

MmProbeAndLockPages

IoBuildSynchronousFsdRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **IoBuildSynchronousFsdRequest** routine allocates and sets up an IRP for a synchronously processed I/O request.

Syntax

```
__drv_aliasesMem PIRP IoBuildSynchronousFsdRequest(
  ULONG            MajorFunction,
  PDEVICE_OBJECT   DeviceObject,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   StartingOffset,
  PKEVENT          Event,
  PIO_STATUS_BLOCK IoStatusBlock
);
```

Parameters

`MajorFunction`

The major function code for the IRP. This code can be IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FLUSH_BUFFERS, or IRP_MJ_SHUTDOWN.

`DeviceObject`

A pointer to the DEVICE_OBJECT structure for the next-lower driver's device object, which represents the target device.

`Buffer`

A pointer to a data buffer. If *MajorFunction* is **IRP_MJ_WRITE**, the buffer contains data to be written. If *MajorFunction* is **IRP_MJ_READ**, the buffer receives data. If *MajorFunction* is **IRP_MJ_FLUSH_BUFFERS** or **IRP_MJ_SHUTDOWN**, this parameter must be **NULL**.

`Length`

The length, in bytes, of the buffer pointed to by *Buffer*. For devices such as disks, this value must be an integer multiple of the sector size. Starting with Windows 8, the sector size can be 4,096 or 512 bytes. In earlier versions of Windows, the sector size is always 512 bytes. This parameter is required for read and write requests, but must be zero for flush and shutdown requests.

`StartingOffset`

A pointer to the offset on the disk, for read and write requests. The units and meaning of this value are driver-specific. This parameter is required for read and write requests, but must be zero for flush and shutdown requests.

`Event`

A pointer to a caller-allocated and initialized event object. The I/O manager sets the event to the Signaled state when a lower-level driver completes the requested operation. After calling IoCallDriver, the driver can wait for the event object.

`IoStatusBlock`

A pointer to a location that receives the I/O status block that is set when the IRP is completed by a lower-level driver.

Return Value

If the operation succeeds, **IoBuildSynchronousFsdRequest** returns a pointer to an initialized IRP structure, with the next-lower driver's I/O stack location set up from the supplied parameters. Otherwise, the routine returns **NULL**.

Remarks

A file system driver (FSD) or other higher-level driver can call **IoBuildSynchronousFsdRequest** to set up IRPs that it synchronously sends to lower-level drivers.

IoBuildSynchronousFsdRequest allocates and sets up an IRP that requests lower-level drivers to perform a synchronous read, write, flush, or shutdown operation. The IRP contains enough information to start the operation.

Lower-level drivers might impose restrictions on parameters supplied to this routine. For example, disk drivers might require that values supplied for *Length* and *StartingOffset* be integer multiples of the device's sector size.

After calling **IoBuildSynchronousFsdRequest** to create a request, the driver must call IoCallDriver to send the request to the next-lower driver. If **IoCallDriver** returns STATUS_PENDING, the driver must wait for the completion of the IRP by calling KeWaitForSingleObject on the given *Event*. Most drivers do not need to set an IoCompletion routine for the IRP.

IRPs that are created by **IoBuildSynchronousFsdRequest** must be completed by a driver's call to IoCompleteRequest. A driver that calls **IoBuildSynchronousFsdRequest** must not call IoFreeIrp, because the I/O manager frees these synchronous IRPs after **IoCompleteRequest** has been called.

IoBuildSynchronousFsdRequest queues the IRPs that it creates to an IRP queue that is specific to the current thread. If the thread exits, the I/O manager cancels the IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	<= APC_LEVEL
DDI compliance rules	ForwardedAtBadIrqlFsdSync, IoBuildSynchronousFsdRequestNoFree, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, PowerIrpDDis, SignalEventInCompletion, HwStorPortProhibitedDDIs

See Also

IO_STACK_LOCATION

IRP

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoCompleteRequest

KeInitializeEvent

KeWaitForSingleObject

IoCallDriver macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoCallDriver** routine, wraps **IofCallDriver** that sends an IRP to the driver associated with a specified device object. For more information, see **IofCallDriver**.

> Do not call **IofCallDriver**. Call **IoCallDriver** instead.

Syntax

```
void IoCallDriver(
   a,
   b
);
```

Parameters

`a`

See **IofCallDriver**.

`b`

See **IofCallDriver**.

Return Value

See **IofCallDriver**.

Remarks

see **IofCallDriver**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

DDI compliance rules	CompleteRequestStatusCheck, CompletionRoutineRegistered, DeleteDevice, ForwardedAtBadIrql, ForwardedAtBadIrqlAllocate, ForwardedAtBadIrqlFsdAsync, ForwardedAtBadIrqlFsdSync, IoAllocateForward, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildFsdForward, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, IoSetCompletionRoutineExCheck, IrpProcessingComplete, LowerDriverReturn, MarkDevicePower, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, PnpIrpCompletion, PowerDownFail, PowerUpFail, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove2, RemoveLockMnSurpriseRemove, RemoveLockQueryMnRemove, TargetRelationNeedsRef, WmiForward, HwStorPortProhibitedDDIs

See Also

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoSetCompletionRoutine

PoCallDriver

IoCancelIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCancelIrp** routine sets the cancel bit in a given IRP and calls the cancel routine for the IRP if there is one.

Syntax

```
BOOLEAN IoCancelIrp(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to the IRP to be canceled.

Return Value

IoCancelIrp returns **TRUE** if the IRP had a cancel routine and this routine was called. Otherwise, it returns **FALSE**. In either case, the IRP's cancel bit is set to **TRUE**. For more information, see Remarks.

Remarks

If the IRP has a cancel routine, **IoCancelIrp** sets the cancel bit and calls the cancel routine.

If *Irp*->**CancelRoutine** is **NULL**, and therefore the IRP is not cancelable, **IoCancelIrp** sets the IRP's cancel bit and returns **FALSE**. The IRP should be canceled at a later time when it becomes cancelable.

If a driver that does not own the IRP calls **IoCancelIrp**, the results are unpredictable. The IRP might be completed with a successful status even though its cancel bit was set.

An intermediate driver should not arbitrarily call **IoCancelIrp** unless that driver created the IRP passed in the call. Otherwise, the intermediate driver might cancel an IRP that some higher-level driver is tracking for purposes of its own.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoSetCancelRoutine

IoCheckLinkShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCheckLinkShareAccess** routine is called by file system drivers (FSDs) or other highest-level drivers to check whether link shared access to a file object is permitted.

Syntax

```
NTSTATUS IoCheckLinkShareAccess(
  ACCESS_MASK        DesiredAccess,
  ULONG              DesiredShareAccess,
  PFILE_OBJECT       FileObject,
  PSHARE_ACCESS      ShareAccess,
  PLINK_SHARE_ACCESS LinkShareAccess,
  ULONG              IoShareAccessFlags
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that indicates the desired type of access to the given file object.

`DesiredShareAccess`

Specifies the desired type of shared access to the file object for the current open request. The value of this parameter is usually the same as the *ShareAccess* parameter that is passed to the file system or highest-level driver by the I/O manager when the open request was made. This value can be zero, or any combination of the following:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

A pointer to the file object for which to check access for the current open request.

`ShareAccess`

A pointer to the common share-access data structure that is associated with *FileObject*. Drivers should treat this structure as opaque.

`LinkShareAccess`

A pointer to the common link share-access data structure (LINK_SHARE_ACCESS) that is associated with *FileObject*. Drivers should treat this structure as opaque.

`IoShareAccessFlags`

A bitmask of these flags:

IO_SHARE_ACCESS_NO_WRITE_PERMISSION (0x80000000) specifies that the user has no write permission for the file. This flag is used to prevent opening a file for exclusive read access when the user does not have

appropriate permissions.

IO_CHECK_SHARE_ACCESS_UPDATE_SHARE_ACCESS (0x00000001) indicates whether the SHARE_ACCESS structure is updated.

Return Value

The **IoCheckLinkShareAccess** routine returns STATUS_SUCCESS if the requester's access to the file object is compatible with the way in which it is currently open. If the request is denied because of a sharing violation, then STATUS_SHARING_VIOLATION is returned.

Remarks

The *ShareAccess* parameter must be locked against other accesses from other threads while this routine is executing. Otherwise the counts can be synchronization.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib

See Also

I/O Manager Routines

IoRemoveLinkShareAccess

IoSetLinkShareAccess

IoUpdateLinkShareAccess

IoCheckShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCheckShareAccess** routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.

Syntax

```
NTSTATUS IoCheckShareAccess(
  ACCESS_MASK   DesiredAccess,
  ULONG         DesiredShareAccess,
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess,
  BOOLEAN       Update
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that indicates the desired type of access to the given *FileObject* for the current open request. Drivers compute the value of this parameter by taking the requested access in the IRP_MJ_CREATE request and then applying SeAccessCheck for each security descriptor to determine the actual access granted. If the granted access is more restrictive than the desired access, then that is an error, and the driver should complete the current IRP with a status of STATUS_ACCESS_DENIED. (Note that **SeAccessCheck** clears the MAXIMUM_ALLOWED bit in the granted access; be sure to not use that bit when comparing desired access to granted access.) The driver then passes the granted access as the value of *DesiredAccess*.

`DesiredShareAccess`

Specifies the desired type of shared access to *FileObject* for the current open request. The value of this parameter is usually the same as the *ShareAccess* passed to the file system or highest-level driver by the I/O manager when the open request was made. This value can be zero, or any combination of the following:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

Pointer to the file object for which to check access for the current open request.

`ShareAccess`

Pointer to the common share-access data structure associated with *FileObject*. Drivers should treat this structure as opaque.

`Update`

Specifies whether to update the share-access status for *FileObject*. A Boolean value of **TRUE** means this routine will update the share access information for the file object if the open request is permitted.

Return Value

IoCheckShareAccess returns STATUS_SUCCESS if the requester's access to the file object is compatible with the way in which it is currently open. If the request is denied because of a sharing violation, then STATUS_SHARING_VIOLATION is returned.

Remarks

IoCheckShareAccess checks a file object open request to determine whether the types of desired and shared accesses specified are compatible with the way in which the file object is currently being accessed by other opens.

File systems maintain state about files through structures called file control blocks (FCBs). The SHARE_ACCESS is a structure describing how the file is currently accessed by all opens. This state is contained in the FCB as part of the open state for each file object. Each file object should have only one share access structure. Other highest-level drivers might call this routine to check the access requested when a file object representing such a driver's device object is opened.

IoCheckShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoCheckShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive2, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

IoCreateFile

IoCreateFileEx

IoGetRelatedDeviceObject

IoRemoveShareAccess

IoSetShareAccess

IoUpdateShareAccess

IoCheckShareAccessEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCheckShareAccessEx** routine is called by file system drivers (FSDs) or other highest-level drivers to check whether shared access to a file object is permitted.

Syntax

```
NTSTATUS IoCheckShareAccessEx(
  ACCESS_MASK   DesiredAccess,
  ULONG         DesiredShareAccess,
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess,
  BOOLEAN       Update,
  PBOOLEAN      WritePermission
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that indicates the desired type of access to the given file object.

`DesiredShareAccess`

Specifies the desired type of shared access to the file object for the current open request. The value of this parameter is usually the same as the *ShareAccess* parameter that is passed to the file system or highest-level driver by the I/O manager when the open request was made. This value can be zero, or any combination of the following:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

A pointer to the file object for which to check access for the current open request.

`ShareAccess`

A pointer to the common share-access data structure that is associated with *FileObject*. Drivers should treat this structure as opaque.

`Update`

Specifies whether to update the share-access status for *FileObject*. A Boolean value of **TRUE** means this routine will update the share access information for the file object if the open request is permitted.

`WritePermission`

Specifies whether the share access has write permission. This value is TRUE if the share has write permission; otherwise, FALSE. If the value is FALSE and the share access has write permission and the caller attempts to take exclusive read access, the write permission is downgraded to FILE_SHARE_READ. This value is NULL if a write permission check is not done on the share access.

Return Value

The **IoCheckShareAccessEx** routine returns STATUS_SUCCESS if the requester's access to the file object is compatible with the way in which it is currently open. If the request is denied because of a sharing violation, then STATUS_SHARING_VIOLATION is returned.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib

See Also

I/O Manager Routines

IoCompleteRequest macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoCompleteRequest** macro indicates that the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O manager.

> **IoCompleteRequest** wraps **IofCompleteRequest**.

Syntax

```
void IoCompleteRequest(
   a,
   b
);
```

Parameters

`a`

Pointer to the IRP to be completed.

`b`

Specifies a system-defined CCHAR constant by which to increment the run-time priority of the original thread that requested the operation. This value is IO_NO_INCREMENT if the original thread requested an operation the driver could complete quickly (so the requesting thread is not compensated for its assumed wait for I/O to be completed) or if the IRP is completed with an error. Otherwise, the set of *PriorityBoost* constants are device-type-specific. See Ntddk.h or Wdm.h for these constants.

Return Value

None

Remarks

When a driver has finished all processing for a given IRP, it calls **IoCompleteRequest**. The I/O manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine for the IRP. If so, each *IoCompletion* routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed a given IRP, the I/O manager returns status to the original requester of the operation. Note that a higher-level driver that sets up a driver-created IRP must supply an *IoCompletion* routine to release the IRP it created.

Never call **IoCompleteRequest** while holding a spin lock. Attempting to complete an IRP while holding a spin lock can cause deadlocks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	CompleteRequest, CompleteRequestStatusCheck, DoubleCompletion, IoAllocateComplete, IoBuildFsdComplete, IoSetCompletionExCompleteIrp, IrpProcessingComplete, MarkIrpPending, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, PnpIrpCompletion, SpinLockSafe, WmiComplete, HwStorPortProhibitedDDIs, SpinLockSafe(storport)

See Also

IoSetCompletionRoutine

IoConnectInterrupt function

1/25/2019 • 2 minutes to read • Edit Online

The **IoConnectInterrupt** routine registers a device driver's InterruptService routine (ISR), so that it will be called when a device interrupts on any of a specified set of processors.

Syntax

```
NTSTATUS IoConnectInterrupt(
  PKINTERRUPT       *InterruptObject,
  PKSERVICE_ROUTINE ServiceRoutine,
  PVOID             ServiceContext,
  PKSPIN_LOCK       SpinLock,
  ULONG             Vector,
  KIRQL             Irql,
  KIRQL             SynchronizeIrql,
  KINTERRUPT_MODE   InterruptMode,
  BOOLEAN           ShareVector,
  KAFFINITY         ProcessorEnableMask,
  BOOLEAN           FloatingSave
);
```

Parameters

`InterruptObject`

Pointer to the address of driver-supplied storage for a pointer to a set of interrupt objects. This pointer must be passed in subsequent calls to KeSynchronizeExecution.

`ServiceRoutine`

Pointer to the entry point for the driver-supplied InterruptService routine.

`ServiceContext`

Pointer to the driver-determined context that will be supplied to the *InterruptService* routine when it is called. The *ServiceContext* area must be in resident memory: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the device driver. See Providing ISR Context Information for details.

`SpinLock`

Pointer to an initialized spin lock, for which the driver supplies the storage, that will be used to synchronize access to driver-determined data shared by other driver routines. This parameter is required if the ISR handles more than one vector or if the driver has more than one ISR. Otherwise, the driver need not allocate storage for an interrupt spin lock and the input pointer is **NULL**.

`Vector`

Specifies the interrupt vector passed in the interrupt resource at the **u.Interrupt.Vector** member of CM_PARTIAL_RESOURCE_DESCRIPTOR.

`Irql`

Specifies the DIRQL passed in the interrupt resource at the **u.Interrupt.Level** member of

CM_PARTIAL_RESOURCE_DESCRIPTOR.

`SynchronizeIrql`

Specifies the DIRQL at which the ISR will run. If the ISR handles more than one interrupt vector or the driver has more than one ISR, this value must be the highest of the *Irql* values passed at **u.Interrupt.Level** in each interrupt resource. Otherwise, the *Irql* and *SynchronizeIrql* values are identical.

`InterruptMode`

Specifies whether the device interrupt is **LevelSensitive** or **Latched**.

`ShareVector`

Specifies whether the interrupt vector is sharable.

`ProcessorEnableMask`

Specifies a **KAFFINITY** value representing the set of processors on which device interrupts can occur in this platform. This value is passed in the interrupt resource at **u.Interrupt.Affinity**.

`FloatingSave`

Specifies whether to save the floating-point stack when the driver's device interrupts. For x86-based and Itanium-based platforms, this value must be set to **FALSE**. For more information about saving floating-point and MMX state, see Using Floating Point or MMX in a WDM Driver.

Return Value

IoConnectInterrupt can return one of the following NTSTATUS values:

Remarks

New drivers should use the IoConnectInterruptEx routine, which is easier to use. Drivers for devices that support message-signaled interrupts (MSI) must use **IoConnectInterruptEx**.

A PnP driver should call **IoConnectInterrupt** as part of device start-up, before it completes the PnP IRP_MN_START_DEVICE request.

When a driver receives an **IRP_MN_START_DEVICE** request, the driver receives raw and translated hardware resources in the **Parameters.StartDevice.AllocatedResources** and **Parameters.StartDevice.AllocatedResourcesTranslated** members of the IRP's IO_STACK_LOCATION structure, respectively. To connect its interrupt, the driver uses the resources at **AllocatedResourcesTranslated.List.PartialResourceList.PartialDescriptors[]**. The driver must scan the array of partial descriptors for resources of type **CmResourceTypeInterrupt**.

If the driver supplies the storage for the *SpinLock*, it must call KeInitializeSpinLock before passing its interrupt spin lock to **IoConnectInterrupt**.

On return from a successful call to **IoConnectInterrupt**, the caller's ISR can be called if interrupts are enabled on the driver's device or if *ShareVector* was set to **TRUE**. Drivers must not enable interrupts until after **IoConnectInterrupt** returns.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive2, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

IO_STACK_LOCATION

IoDisconnectInterrupt

KeInitializeSpinLock

KeSynchronizeExecution

IoConnectInterruptEx function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibIoConnectInterruptEx function.

```
#define IoConnectInterruptEx WdmlibIoConnectInterruptEx
```

Syntax

```
NTSTATUS IoConnectInterruptEx(
  PIO_CONNECT_INTERRUPT_PARAMETERS Parameters
);
```

Parameters

`Parameters`

For more information, see the WdmlibIoConnectInterruptEx function.

Return Value

For more information, see the WdmlibIoConnectInterruptEx function.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system. Drivers that must also work on Windows 2000, Windows XP, or Windows Server 2003 can instead link to Iointex.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

WdmlibIoConnectInterruptEx

IoCopyCurrentIrpStackLocationToNext function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCopyCurrentIrpStackLocationToNext** routine copies the IRP stack parameters from the current I/O stack location to the stack location of the next-lower driver.

Syntax

```
void IoCopyCurrentIrpStackLocationToNext(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to the IRP.

Return Value

None

Remarks

A driver calls **IoCopyCurrentIrpStackLocationToNext** to copy the IRP parameters from its stack location to the next-lower driver's stack location.

After calling this routine, a driver typically sets an I/O completion routine with IoSetCompletionRoutine before passing the IRP to the next-lower driver with IoCallDriver. Drivers that pass on their IRP parameters but do not set an I/O completion routine should call IoSkipCurrentIrpStackLocation instead of this routine.

If your driver calls **IoSkipCurrentIrpStackLocation**, be careful not to modify the IO_STACK_LOCATION structure in a way that could unintentionally affect the lower driver or the system's behavior with respect to that driver. In particular, your driver should not modify the **IO_STACK_LOCATION** structure's **Parameters** union, and should not call the IoMarkIrpPending routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<=DISPATCH_LEVEL"

See Also

IO_STACK_LOCATION

IoCallDriver

IoMarkIrpPending

IoSetCompletionRoutine

IoSkipCurrentIrpStackLocation

IoCreateDevice function

1/25/2019 • 3 minutes to read • Edit Online

The **IoCreateDevice** routine creates a device object for use by a driver.

Syntax

```
NTSTATUS IoCreateDevice(
  PDRIVER_OBJECT  DriverObject,
  ULONG           DeviceExtensionSize,
  PUNICODE_STRING DeviceName,
  DEVICE_TYPE     DeviceType,
  ULONG           DeviceCharacteristics,
  BOOLEAN         Exclusive,
  PDEVICE_OBJECT  *DeviceObject
);
```

Parameters

`DriverObject`

Pointer to the driver object for the caller. Each driver receives a pointer to its driver object in a parameter to its DriverEntry routine. WDM function and filter drivers also receive a driver object pointer in their AddDevice routines.

`DeviceExtensionSize`

Specifies the driver-determined number of bytes to be allocated for the device extension of the device object. The internal structure of the device extension is driver-defined.

`DeviceName`

Optionally points to a buffer containing a null-terminated Unicode string that names the device object. The string must be a full path name. WDM filter and function drivers do not name their device objects. For more information, see Named Device Objects.

> [!ALERT] If a device name is not supplied (that is, *DeviceName* is **NULL**), the device object created by **IoCreateDevice** will not (and cannot) have a discretionary access control list (DACL) associated with it. For additional information, see Security Descriptors.

`DeviceType`

Specifies one of the system-defined FILE_DEVICE_*XXX* constants that indicate the type of device (such as FILE_DEVICE_DISK or FILE_DEVICE_KEYBOARD) or a vendor-defined value for a new type of device. For more information, see Specifying Device Types.

`DeviceCharacteristics`

Specifies one or more system-defined constants, ORed together, that provide additional information about the driver's device. For a list of possible device characteristics, see DEVICE_OBJECT. For more information about how to specify device characteristics, see Specifying Device Characteristics. Most drivers specify FILE_DEVICE_SECURE_OPEN for this parameter.

`Exclusive`

Specifies if the device object represents an exclusive device. Most drivers set this value to **FALSE**. For more information about exclusive access, see Specifying Exclusive Access to Device Objects.

`DeviceObject`

Pointer to a variable that receives a pointer to the newly created DEVICE_OBJECT structure. The **DEVICE_OBJECT** structure is allocated from nonpaged pool.

Return Value

IoCreateDevice returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. A partial list of the failure codes returned by this function includes:

- **STATUS_INSUFFICIENT_RESOURCES**
- **STATUS_OBJECT_NAME_COLLISION**

Remarks

IoCreateDevice creates a device object and returns a pointer to the object. The caller is responsible for deleting the object when it is no longer needed by calling IoDeleteDevice.

IoCreateDevice can only be used to create an unnamed device object, or a named device object for which a security descriptor is set by an INF file. Otherwise, drivers must use IoCreateDeviceSecure to create named device objects. For more information, see Creating a Device Object. The caller is responsible for setting certain members of the returned device object. For more information, see Initializing a Device Object and the device-type-specific documentation for your device.

Be careful to specify the *DeviceType* and *DeviceCharacteristics* values in the correct parameters. Both parameters use system-defined FILE_*XXX* constants and some driver writers specify the values in the wrong parameters by mistake.

A remote file system that creates a named device object for a network redirector, and that registers using FsRtlRegisterUncProvider, must specify FILE_REMOTE_DEVICE as one of the options in the *DeviceCharacteristics* parameter of **IoCreateDevice**.

Device objects for disks, tapes, CD-ROMs, and RAM disks are given a Volume Parameter Block (VPB) that is initialized to indicate that the volume has never been mounted on the device.

If a driver's call to **IoCreateDevice** returns an error, the driver should release any resources that it allocated for that device.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	AddDevice, CheckDeviceObjectFlags, IrqlIoPassive1, MiniportOnlyWdmDevice, HwStorPortProhibitedDDIs

See Also

DEVICE_OBJECT

FsRtlRegisterUncProvider

IoAttachDevice

IoAttachDeviceToDeviceStack

IoCreateDeviceSecure

IoCreateSymbolicLink

IoDeleteDevice

IoCreateFile function

1/25/2019 • 15 minutes to read • Edit Online

The **IoCreateFile** routine either causes a new file or directory to be created, or it opens an existing file, device, directory, or volume, giving the caller a handle for the file object.

Syntax

```
NTSTATUS IoCreateFile(
  PHANDLE            FileHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PIO_STATUS_BLOCK   IoStatusBlock,
  PLARGE_INTEGER     AllocationSize,
  ULONG              FileAttributes,
  ULONG              ShareAccess,
  ULONG              Disposition,
  ULONG              CreateOptions,
  PVOID              EaBuffer,
  ULONG              EaLength,
  CREATE_FILE_TYPE   CreateFileType,
  PVOID              InternalParameters,
  ULONG              Options
);
```

Parameters

`FileHandle`

A pointer to a variable that receives the file handle if the call is successful. The driver must close the handle with ZwClose once the handle is no longer in use.

`DesiredAccess`

Specifies the ACCESS_MASK value that represents the type of access that the caller requires to the file or directory. See ZwCreateFile for a description of the possible values for this parameter.

`ObjectAttributes`

A pointer to a structure that specifies the object's attributes, which has already been initialized with InitializeObjectAttributes. If the caller is not running in the system process context, it must set the OBJ_KERNEL_HANDLE attribute for *ObjectAttributes*.

`IoStatusBlock`

A pointer to a variable that receives the final completion status and information about the requested operation. On return from **IoCreateFile**, the **Information** member contains one of the following values:

- FILE_CREATED
- FILE_OPENED
- FILE_OVERWRITTEN
- FILE_SUPERSEDED
- FILE_EXISTS
- FILE_DOES_NOT_EXIST

`AllocationSize`

Optionally specifies the initial allocation size in bytes for the file. A nonzero value has no effect unless the file is being created, overwritten, or superseded.

`FileAttributes`

Explicitly specified attributes are applied only when the file is created, superseded, or, in some cases, overwritten. By default, this value is FILE_ATTRIBUTE_NORMAL, which can be overridden by an ORed combination of one or more FILE_ATTRIBUTE_*XXX* flags, which are defined in Wdm.h. For a list of flags that can be used with **IoCreateFile**, see CreateFile in the Microsoft Windows SDK documentation.

`ShareAccess`

Specifies the type of share access that the caller would like to the file, as zero, or as one or a combination of the following:

SHAREACCESS FLAGS	MEANING
FILE_SHARE_READ	The file can be opened for read access by other threads' calls to **IoCreateFile**.
FILE_SHARE_WRITE	The file can be opened for write access by other threads' calls to **IoCreateFile**.
FILE_SHARE_DELETE	The file can be opened for delete access by other threads' calls to **IoCreateFile**.

Device and intermediate drivers usually set *ShareAccess* to zero, which gives the caller exclusive access to the open file.

`Disposition`

Specifies what to do, depending on whether the file already exists, as one of the following:

DISPOSITION VALUES	MEANING
FILE_SUPERSEDE	If the file already exists, replace it with the given file. If it does not, create the given file.
FILE_CREATE	If the file already exists, fail the request and do not create or open the given file. If it does not, create the given file.
FILE_OPEN	If the file already exists, open it instead of creating a new file. If it does not, fail the request and do not create a new file.
FILE_OPEN_IF	If the file already exists, open it. If it does not, create the given file.
FILE_OVERWRITE	If the file already exists, open it and overwrite it. If it does not, fail the request.
FILE_OVERWRITE_IF	If the file already exists, open it and overwrite it. If it does not, create the given file.

`CreateOptions`

Specifies the options to be applied when creating or opening the file, as a compatible combination of the following

flags:

CREATEOPTIONS FLAGS	MEANING
FILE_DIRECTORY_FILE	The file being created or opened is a directory file. With this flag, the *Disposition* parameter must be set to one of FILE_CREATE, FILE_OPEN, or FILE_OPEN_IF. With this flag, other compatible *CreateOptions* flags include only the following: FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT, and FILE_OPEN_BY_FILE_ID.
FILE_NON_DIRECTORY_FILE	The file being opened must not be a directory file or this call will fail. The file object being opened can represent a data file, a logical, virtual, or physical device, or a volume.
FILE_WRITE_THROUGH	System services, file system drivers (FSDs), and drivers that write data to the file must actually transfer the data into the file before any requested write operation is considered complete.
FILE_SEQUENTIAL_ONLY	All accesses to the file will be sequential.
FILE_RANDOM_ACCESS	Accesses to the file can be random, so no sequential read-ahead operations should be performed on the file by FSDs or the system.
FILE_NO_INTERMEDIATE_BUFFERING	The file cannot be cached or buffered in a driver's internal buffers. This flag is incompatible with the *DesiredAccess* FILE_APPEND_DATA flag.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *DesiredAccess* SYNCHRONIZE flag also must be set.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the file are performed synchronously. Waits in the system to synchronize I/O queuing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *DesiredAccess* SYNCHRONIZE flag also must be set.
FILE_CREATE_TREE_CONNECTION	Create a tree connection for this file in order to open it over the network. This flag is irrelevant to device and intermediate drivers.
FILE_COMPLETE_IF_OPLOCKED	Complete this operation immediately with an alternate success code if the target file is oplocked, rather than blocking the caller's thread. If the file is oplocked, another caller already has access to the file over the network. This flag is irrelevant to device and intermediate drivers.
FILE_NO_EA_KNOWLEDGE	If the extended attributes on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, fail this request because the caller does not understand how to deal with EAs. Device and intermediate drivers can ignore this flag.

FILE_OPEN_REPARSE_POINT	Open a file with a reparse point and bypass normal reparse point processing for the file. For more information, see the following Remarks section.
FILE_DELETE_ON_CLOSE	Delete the file when the last handle to it is passed to **ZwClose**.
FILE_OPEN_BY_FILE_ID	The file name specified in the *ObjectAttributes* parameter includes the 8-byte file reference number for the file. This number is assigned by the file system and is file-system-specific. If the file is a reparse point, the file name also includes the name of a device. Note: The FAT file system does not support FILE_OPEN_BY_FILE_ID.
FILE_OPEN_FOR_BACKUP_INTENT	The file is being opened for backup intent, hence, the system should check for certain access rights and grant the caller the appropriate accesses to the file before checking the input *DesiredAccess* against the file's security descriptor. This flag is irrelevant to device and intermediate drivers.
FILE_OPEN_REQUIRING_OPLOCK	The file is being opened and an opportunistic lock (oplock) on the file is being requested as a single atomic operation. The file system checks for oplocks before it performs the create operation, and the create operation will fail with a return code of STATUS_CANNOT_BREAK_OPLOCK if the create operation would break an existing oplock. **Note** The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2, and later Windows operating systems.
FILE_RESERVE_OPFILTER	This flag allows an application to request a filter opportunistic lock (oplock) to prevent other applications from getting share violations. If there are already open handles, the create request will fail with STATUS_OPLOCK_NOT_GRANTED. For more information, see the following Remarks section.

`EaBuffer`

For device and intermediate drivers, this parameter must be a **NULL** pointer.

`EaLength`

For device and intermediate drivers, this parameter must be zero.

`CreateFileType`

Drivers must set this parameter to **CreateFileTypeNone**.

`InternalParameters`

Drivers must set this parameter to **NULL**.

`Options`

Specifies options to be used during the creation of the create request. These options can be from the following list:

OPTIONS FLAGS	MEANING

IO_NO_PARAMETER_CHECKING	Indicates that the parameters for this call should not be validated before attempting to issue the create request. Driver writers should use this flag with caution as certain invalid parameters can cause a system failure. For more information, see Remarks.
IO_FORCE_ACCESS_CHECK	Indicates that the I/O manager must check the operation against the file's security descriptor. For more information, see Remarks.
IO_STOP_ON_SYMLINK	The I/O manager or the file system will return STATUS_STOPPED_ON_SYMLINK if a symbolic link is encountered while opening or creating the file.
IO_OPEN_TARGET_DIRECTORY	Open the file's parent directory.

Return Value

IoCreateFile either returns STATUS_SUCCESS or an appropriate error status. If it returns an error status, the caller can find additional information about the cause of the failure by checking the *IoStatusBlock*.

Remarks

> **NOTE**
> The **IoCreateFileEx** routine is similar to the **IoCreateFile** routine but provides greater functionality than the **IoCreateFile** routine, including access to extra create parameters (ECPs), device objects, and transaction information.

The handle obtained by **IoCreateFile** can be used by subsequent calls to manipulate data within the file or the file object's state or attributes.

There are two alternate ways to specify the name of the file to be created or opened with **IoCreateFile**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *ObjectAttributes*
2. As pathname relative to the directory file represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*

Certain *DesiredAccess* flags and combinations of flags have the following effects:

- For a caller to synchronize an I/O completion by waiting for the returned *FileHandle*, the SYNCHRONIZE flag must be set. Otherwise, a caller that is a device or intermediate driver must synchronize an I/O completion by using an event object.
- If only the FILE_APPEND_DATA and SYNCHRONIZE flags are set, the caller can write only to the end of the file, and any offset information about writes to the file is ignored. However, the file will automatically be extended as necessary for this type of write operation.
- Setting the FILE_WRITE_DATA flag for a file also allows writes beyond the end of the file to occur. The file is automatically extended for this type of write, as well.
- If only the FILE_EXECUTE and SYNCHRONIZE flags are set, the caller cannot directly read or write any data in the file using the returned *FileHandle*: that is, all operations on the file occur through the system pager in response to instruction and data accesses. Device and intermediate drivers should not set the FILE_EXECUTE flag in *DesiredAccess*.

The *ShareAccess* parameter determines whether separate threads can access the same file, possibly simultaneously.

Provided that both file openers have the privilege to access a file in the specified manner, the file can be successfully opened and shared. If the original caller of **IoCreateFile** does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, no other open operations can be performed on the file: that is, the original caller is given exclusive access to the file.

In order for a shared file to be successfully opened, the requested *DesiredAccess* to the file must be compatible with both the *DesiredAccess* and *ShareAccess* specifications of all preceding opens that have not yet been released with **ZwClose**. That is, the *DesiredAccess* specified to **IoCreateFile** for a given file must not conflict with the accesses that other openers of the file have disallowed.

The *Disposition* value FILE_SUPERSEDE requires that the caller have DELETE access to a existing file object. If so, a successful call to **IoCreateFile** with FILE_SUPERSEDE on an existing file effectively deletes that file, and then recreates it. This implies that, if the file has already been opened by another thread, it opened the file by specifying a *ShareAccess *parameter with the FILE_SHARE_DELETE flag set. Note that this type of disposition is consistent with the POSIX style of overwriting files.

The *Disposition* values FILE_OVERWRITE_IF and FILE_SUPERSEDE are similar. If **IoCreateFile** is called with a existing file and either of these *Disposition* values, the file will be replaced.

Overwriting a file is semantically equivalent to a supersede operation, except for the following:

- The caller must have write access to the file, rather than delete access. This implies that, if the file has already been opened by another thread, it opened the file with the FILE_SHARE_WRITE flag set in the input *ShareAccess*.
- The specified file attributes are logically ORed with those already on the file. This implies that, if the file has already been opened by another thread, a subsequent caller of **IoCreateFile** cannot disable existing *FileAttributes* flags but can enable additional flags for the same file.

The *CreateOptions* FILE_DIRECTORY_FILE value specifies that the file to be created or opened is a directory file. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the given file to be opened is not a directory file, or if the caller specified an inconsistent *CreateOptions* or *Disposition* value, the call to **IoCreateFile** will fail.

The *CreateOptions* FILE_NO_INTERMEDIATE_BUFFERING flag prevents the file system from performing any intermediate buffering on behalf of the caller. Specifying this value places certain restrictions on the caller's parameters to the **Zw*Xxx*File** routines, including the following:

- Any optional *ByteOffset* passed to ZwReadFile or ZwWriteFile must be an integral of the sector size.
- The *Length* passed to **ZwReadFile** or **ZwWriteFile**, must be an integral of the sector size. Note that specifying a read operation to a buffer whose length is exactly the sector size might result in a lesser number of significant bytes being transferred to that buffer if the end of the file was reached during the transfer.
- Buffers must be aligned in accordance with the alignment requirement of the underlying device. This information can be obtained by calling **IoCreateFile** to get a handle for the file object that represents the physical device, and, then, calling ZwQueryInformationFile with that handle. For a list of the system FILE_*XXX*_ALIGNMENT values, see DEVICE_OBJECT.
- Calls to ZwSetInformationFile with the *FileInformationClass* parameter set to **FilePositionInformation** must specify an offset that is an integral of the sector size.

The *CreateOptions* FILE_SYNCHRONOUS_IO_ALERT and FILE_SYNCHRONOUS_IO_NONALERT, which are mutually exclusive as their names suggest, specify that all I/O operations on the file are to be synchronous as long as they occur through the file object referred to by the returned *FileHandle*. All I/O on such a file is serialized across all threads using the returned handle. With either of these *CreateOptions*, the *DesiredAccess* SYNCHRONIZE flag must be set so that the I/O manager will use the file object as a synchronization object. With

either of these *CreateOptions* set, the I/O manager maintains the "file position context" for the file object, an internal, current file position offset. This offset can be used in calls to **ZwReadFile** and **ZwWriteFile**. Its position also can be queried or set with **ZwQueryInformationFile** and **ZwSetInformationFile**.

If the *CreateOptions* FILE_OPEN_REPARSE_POINT flag is not specified and **IoCreateFile** attempts to open a file with a reparse point, normal reparse point processing occurs for the file. If, on the other hand, the FILE_OPEN_REPARSE_POINT flag is specified, normal reparse processing does not occur and **IoCreateFile** attempts to directly open the reparse point file. In either case, if the open operation was successful, **IoCreateFile** returns STATUS_SUCCESS; otherwise, the routine returns an NTSTATUS error code. **IoCreateFile** never returns STATUS_REPARSE.

The *CreateOptions* FILE_OPEN_REQUIRING_OPLOCK flag eliminates the time between when you open the file and request an oplock that could potentially enable a third party to open the file and get a sharing violation. An application can use the FILE_OPEN_REQUIRING_OPLOCK flag on **IoCreateFile** and then request any oplock. This ensures that an oplock owner will be notified of any later open request that causes a sharing violation.

In Windows 7, if other handles exist on the file when an application uses the FILE_OPEN_REQUIRING_OPLOCK flag, the create operation will fail with STATUS_OPLOCK_NOT_GRANTED. This restriction no longer exists starting with Windows 8.

If this create operation would break an oplock that already exists on the file, then setting the FILE_OPEN_REQUIRING_OPLOCK flag will cause the create operation to fail with STATUS_CANNOT_BREAK_OPLOCK. The existing oplock will not be broken by this create operation.

An application that uses the FILE_OPEN_REQUIRING_OPLOCK flag must request an oplock after this call succeeds, or all subsequent attempts to open the file will be blocked without the benefit of normal oplock processing. Similarly, if this call succeeds but the subsequent oplock request fails, an application that uses this flag must close its handle after it detects that the oplock request has failed.

> **NOTE**
>
> The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2 and later versions of Windows. The Microsoft file systems that implement this flag in Windows 7 are NTFS, FAT, and exFAT.

The *CreateOptions* flag, FILE_RESERVE_OPFILTER, allows an application to request a level 1, batch, or filter oplock to prevent other applications from getting share violations. However, FILE_RESERVE_OPFILTER is only useful for filter oplocks. To use it, you must follow these steps:

1. Issue a create request with *CreateOptions* of FILE_RESERVE_OPFILTER, *DesiredAccess* of exactly FILE_READ_ATTRIBUTES, and *ShareAccess* of exactly FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE.
   - If there are already open handles, the create request fails with STATUS_OPLOCK_NOT_GRANTED, and the next requested oplock also fails.
   - If you open with more access or less sharing will also cause a failure of STATUS_OPLOCK_NOT_GRANTED.
2. If the create request succeeds, request an oplock.
3. Open another handle to the file to do I/O.

Step 3 makes this practical only for filter oplocks. The handle opened in step 3 can have a *DesiredAccess* that contains a maximum of FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | FILE_READ_DATA | FILE_READ_EA | FILE_EXECUTE | SYNCHRONIZE | READ_CONTROL and still not break a filter oplock. However, any *DesiredAccess* greater than FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | SYNCHRONIZE will break a level 1 or batch oplock and make the FILE_RESERVE_OPFILTER flag useless for those oplock types.

The *Options* IO_NO_PARAMETER_CHECKING flag can be useful if a driver is issuing a kernel-mode create

request on behalf of an operation initiated by a user-mode application. Because the request occurs within a user-mode context, the I/O manager, by default, probes the supplied parameter values, which can cause an access violation if the parameters are kernel-mode addresses. This flag enables the caller to override this default behavior and avoid the access violation.

For create requests originating in user mode, if the driver sets both IO_NO_PARAMETER_CHECKING and IO_FORCE_ACCESS_CHECK in the *Options* parameter of **IoCreateFile** then it should also set OBJ_FORCE_ACCESS_CHECK in the *ObjectAttributes* parameter. For info on this flag, see the **Attributes** member of OBJECT_ATTRIBUTES.

NTFS is the only Microsoft file system that implements FILE_RESERVE_OPFILTER.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE attribute for the *ObjectAttributes* parameter of **IoCreateFile**. This restricts the use of the handle returned by **IoCreateFile** to processes running only in kernel mode. Otherwise, the handle can be accessed by the process in whose context the driver is running. Drivers can call InitializeObjectAttributes to set the OBJ_KERNEL_HANDLE attribute as follows.

```
InitializeObjectAttributes(&amp;ObjectAttributes, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
```

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

ZwCreateFile

IoCreateNotificationEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCreateNotificationEvent** routine creates or opens a named notification event used to notify one or more threads of execution that an event has occurred.

Syntax

```
PKEVENT IoCreateNotificationEvent(
  PUNICODE_STRING EventName,
  PHANDLE         EventHandle
);
```

Parameters

`EventName`

Pointer to a buffer containing a null-terminated Unicode string that names the event.

`EventHandle`

Pointer to a location in which to return a handle for the event object. In Windows Server 2003 and later versions of Windows, the returned handle is a kernel handle.

Return Value

IoCreateNotificationEvent returns a pointer to the created or opened event object or **NULL** if the event object could not be created or opened.

Remarks

IoCreateNotificationEvent creates and opens the event object if it does not already exist. **IoCreateNotificationEvent** sets the state of a new notification event to Signaled. If the event object already exists, **IoCreateNotificationEvent** just opens the event object.

When a notification event is set to the Signaled state it remains in that state until it is explicitly cleared.

Notification events, like synchronization events, are used to coordinate execution. Unlike a synchronization event, a notification event is not auto-resetting. Once a notification event is in the Signaled state, it remains in that state until it is explicitly reset (with a call to KeClearEvent or KeResetEvent).

To synchronize on a notification event:

1. Open the notification event with **IoCreateNotificationEvent**. Identify the event with the *EventName* string.
2. Wait for the event to be signaled by calling KeWaitForSingleObject with the PKEVENT returned by **IoCreateNotificationEvent**. More than one thread of execution can wait for a given notification event. To poll instead of stall, specify a *Timeout* of zero to **KeWaitForSingleObject**.
3. Close the handle to the notification event with ZwClose when access to the event is no longer needed.

Sharing event objects between user mode and kernel mode requires care. There are two main methods for sharing event objects:

- The user-mode application creates the event object and passes a handle to the object to the driver by sending an IOCTL to the driver. The driver must handle the IOCTL in the context of the process that created the event object and must validate the handle by calling ObReferenceObjectByHandle. This method is the recommended method for sharing event objects between user and kernel modes.
- The driver creates a named event object in the \BaseNamedObjects object directory. You can open a kernel-mode event named \BaseNamedObjects<i>Xxx in user mode under the name *Xxx*. Note that security settings can prevent an application from opening the event. For more information, see the OpenEvent Fails in a Non-Administrator Account KB article. The \BaseNamedObjects object directory is not created until the Microsoft Win32 subsystem initializes, so drivers that are loaded at boot time cannot create event objects in the \BaseNamedObjects directory in their DriverEntry routines.

For more information about events, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, HwStorPortProhibitedDDIs

See Also

IoCreateSynchronizationEvent

KeClearEvent

KeResetEvent

KeSetEvent

KeWaitForSingleObject

RtlInitUnicodeString

ZwClose

IoCreateSymbolicLink function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCreateSymbolicLink** routine sets up a symbolic link between a device object name and a user-visible name for the device.

Syntax

```
NTSTATUS IoCreateSymbolicLink(
  PUNICODE_STRING SymbolicLinkName,
  PUNICODE_STRING DeviceName
);
```

Parameters

`SymbolicLinkName`

Pointer to a buffered Unicode string that is the user-visible name.

`DeviceName`

Pointer to a buffered Unicode string that is the name of the driver-created device object.

Return Value

IoCreateSymbolicLink returns STATUS_SUCCESS if the symbolic link object was created.

Remarks

WDM drivers do not name device objects and therefore should not use this routine. Instead, a WDM driver should call IoRegisterDeviceInterface to set up a symbolic link.

For more information about when to use **IoCreateSymbolicLink**, see Named Device Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	IrqlIoPassive3, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoAssignArcName

IoCreateUnprotectedSymbolicLink

IoDeleteSymbolicLink

IoRegisterDeviceInterface

IoCreateSynchronizationEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCreateSynchronizationEvent** routine creates or opens a named synchronization event for use in serialization of access to hardware between two otherwise unrelated drivers.

Syntax

```
PKEVENT IoCreateSynchronizationEvent(
  PUNICODE_STRING EventName,
  PHANDLE         EventHandle
);
```

Parameters

`EventName`

Pointer to a buffer containing a null-terminated Unicode string that names the event.

`EventHandle`

Pointer to a location in which to return a handle for the event object. In Windows Server 2003 and later versions of Windows, the returned handle is a kernel handle.

Return Value

IoCreateSynchronizationEvent returns a pointer to the created or opened event object or **NULL** if the event object could not be created or opened.

Remarks

The event object is created if it does not already exist. **IoCreateSynchronizationEvent** sets the state of a new synchronization event to Signaled. If the event object already exists, it is simply opened. The pair of drivers that use a synchronization event call KeWaitForSingleObject with the PKEVENT pointer returned by this routine.

When a synchronization event is set to the Signaled state, a single thread of execution that was waiting for the event is released, and the event is automatically reset to the Not-Signaled state.

To release the event, a driver calls ZwClose with the event handle.

Sharing event objects between user mode and kernel mode requires care. There are two main methods for sharing event objects:

- The user-mode application creates the event object and passes a handle to the object to the driver by sending an IOCTL to the driver. The driver must handle the IOCTL in the context of the process that created the event object and must validate the handle by calling ObReferenceObjectByHandle. This method is the recommended method for sharing event objects between user and kernel modes.
- The driver creates a named event object in the \BaseNamedObjects object directory. You can open a kernel-mode event named \BaseNamedObjects<i>Xxx in user mode under the name *Xxx*. Note that security settings can prevent an application from opening the event. For more information, see the OpenEvent Fails in a Non-Administrator Account KB article. The \BaseNamedObjects object directory is not created until the Microsoft

Win32 subsystem initializes, so drivers that are loaded at boot time cannot create event objects in the \BaseNamedObjects directory in their DriverEntry routines

For more information about events, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoCreateNotificationEvent

KeWaitForSingleObject

RtlInitUnicodeString

ZwClose

IoCreateSystemThread function

1/25/2019 • 3 minutes to read • Edit Online

The **IoCreateSystemThread** routine creates a system thread that executes in kernel mode, and supplies a handle for the thread.

Syntax

```
NTSTATUS IoCreateSystemThread(
  PVOID              IoObject,
  PHANDLE            ThreadHandle,
  ULONG              DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  HANDLE             ProcessHandle,
  PCLIENT_ID         ClientId,
  PKSTART_ROUTINE    StartRoutine,
  PVOID              StartContext
);
```

Parameters

`IoObject`

A pointer to the DEVICE_OBJECT or DRIVER_OBJECT to associate with the created thread. **IoCreateSystemThread** takes a counted reference to this object. The I/O manager later releases this reference when the thread exits. For more information, see Remarks.

`ThreadHandle`

A pointer to a variable to which the routine writes the kernel handle for the created thread. When the handle is no longer needed, the driver must close the handle by calling the ZwClose routine.

`DesiredAccess`

The ACCESS_MASK value that represents the types of access the caller requests to the created thread.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the thread object's attributes. The OBJ_PERMANENT, OBJ_EXCLUSIVE, and OBJ_OPENIF attributes are not valid attributes for a thread object. If the caller is not running in the system process context, it must set the OBJ_KERNEL_HANDLE attribute in the **OBJECT_ATTRIBUTES** structure.

`ProcessHandle`

An open handle for the process in whose address space the created thread is to run. The caller's thread must have PROCESS_CREATE_THREAD access to this process. If this parameter is **NULL**, the thread will be created in the initial system process. This parameter should be **NULL** for a driver-created thread. Use the **NtCurrentProcess** macro, defined in the Wdm.h header file, to specify the current process.

`ClientId`

A pointer to a structure to which the routine writes the client identifier for the created thread. This parameter should be **NULL** for a driver-created thread.

`StartRoutine`

A pointer to a ThreadStart routine that is the entry point for the created thread.

`StartContext`

A context pointer that is passed as the *StartContext* parameter to the ThreadStart routine when the created thread starts to run.

Return Value

IoCreateSystemThread returns STATUS_SUCCESS if the new thread was successfully created. Possible return values include the following error status codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_HANDLE	*ProcessHandle* is not a valid process handle.
STATUS_PROCESS_IS_TERMINATING	The process specified by *ProcessHandle* is terminating.
STATUS_INSUFFICIENT_RESOURCES	Insufficient system resources are available to perform the requested operation.

Remarks

Starting with Windows 8, a driver can call **IoCreateSystemThread** to create a device-dedicated thread. This routine creates a new system thread that has no thread environment block (TEB) or user-mode context, and runs only in kernel mode.

Typically, the driver calls **IoCreateSystemThread** either when it starts the device, or when the driver's *Dispatch*Xxx routines start to receive I/O requests. For example, a driver might call this routine to create a thread when a *Dispatch*Xxx routine receives an asynchronous device control request.

If the *ProcessHandle* parameter is **NULL**, the created thread is associated with the system process. Such a thread continues running until either the system is shut down or the thread exits.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE attribute for the *ObjectAttributes* parameter of **IoCreateSystemThread**. This attribute restricts the use of the handle returned by **IoCreateSystemThread** to processes running in kernel mode. Otherwise, the thread handle could be accessed by the process in whose context the driver is running. Drivers can call the InitializeObjectAttributes macro to set the OBJ_KERNEL_HANDLE attribute in the object attributes, as shown in the following code example.

```
OBJECT_ATTRIBUTES ObjectAttributes;

InitializeObjectAttributes(&ObjectAttributes, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
```

IoCreateSystemThread is similar to the PsCreateSystemThread routine, but has an additional parameter, *IoObject*, which is a pointer to the caller's driver object or device object. **IoCreateSystemThread** uses this parameter to ensure that the driver cannot unload while the created thread exists. Before scheduling *StartRoutine* to run in this thread, **IoCreateSystemThread** takes a counted reference to the *IoObject* object. The I/O manager releases this reference after the created thread exits. Thus, this object persists for the lifetime of the created thread.

In contrast to a system thread that is created by the PsCreateSystemThread routine, a thread created by **IoCreateSystemThread** does not call the PsTerminateSystemThread routine to terminate itself. Instead, the I/O manager calls **PsTerminateSystemThread** on behalf of the created thread when the thread exits.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ACCESS_MASK

DEVICE_OBJECT

DRIVER_OBJECT

InitializeObjectAttributes

OBJECT_ATTRIBUTES

PsCreateSystemThread

PsTerminateSystemThread

ThreadStart

ZwClose

IoCreateUnprotectedSymbolicLink function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCreateUnprotectedSymbolicLink** routine sets up an unprotected symbolic link between a device object name and a corresponding Win32-visible name.

Syntax

```
NTSTATUS IoCreateUnprotectedSymbolicLink(
  PUNICODE_STRING SymbolicLinkName,
  PUNICODE_STRING DeviceName
);
```

Parameters

`SymbolicLinkName`

Supplies the symbolic link name as a Unicode string.

`DeviceName`

Supplies the name of the device object to which the symbolic link name refers.

Return Value

IoCreateUnprotectedSymbolicLink returns the final status of the operation.

Remarks

WDM drivers do not name device objects and therefore should not use this routine. Instead, a WDM driver should call IoRegisterDeviceInterface to set up a symbolic link.

IoCreateUnprotectedSymbolicLink can be used by drivers if the user needs to be able to manipulate the symbolic link. For example, the parallel and serial drivers create unprotected symbolic links for LPTx and COMx, so that users can manipulate and reassign them by using the MODE command.

In general, drivers should call this routine instead of IoCreateSymbolicLink if a protected subsystem lets end users change what a named device references as, for example, when using LPT1 to access a network printer.

For more information about when to use **IoCreateSymbolicLink**, see Named Device Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoAssignArcName

IoCreateSymbolicLink

IoDeleteSymbolicLink

IoRegisterDeviceInterface

IoCsqInitialize function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqInitialize** routine initializes the driver's cancel-safe IRP queue dispatch table.

Syntax

```
NTSTATUS IoCsqInitialize(
  PIO_CSQ                      Csq,
  PIO_CSQ_INSERT_IRP           CsqInsertIrp,
  PIO_CSQ_REMOVE_IRP           CsqRemoveIrp,
  PIO_CSQ_PEEK_NEXT_IRP        CsqPeekNextIrp,
  PIO_CSQ_ACQUIRE_LOCK         CsqAcquireLock,
  PIO_CSQ_RELEASE_LOCK         CsqReleaseLock,
  PIO_CSQ_COMPLETE_CANCELED_IRP CsqCompleteCanceledIrp
);
```

Parameters

`Csq`

Pointer to the IO_CSQ structure to be initialized by **IoCsqInitialize**.

`CsqInsertIrp`

Pointer to the driver-defined CsqInsertIrp function for the driver's cancel-safe IRP queue.

`CsqRemoveIrp`

Pointer to the driver-defined CsqRemoveIrp function for the driver's cancel-safe IRP queue.

`CsqPeekNextIrp`

Pointer to the driver-defined CsqPeekNextIrp function for the driver's cancel-safe IRP queue.

`CsqAcquireLock`

Pointer to the driver-defined CsqAcquireLock function for the driver's cancel-safe IRP queue.

`CsqReleaseLock`

Pointer to the driver-defined CsqReleaseLock function for the driver's cancel-safe IRP queue.

`CsqCompleteCanceledIrp`

Pointer to the driver-defined CsqCompleteCanceledIrp function for the driver's cancel-safe IRP queue.

Return Value

This routine returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

The **IoCsqInitialize** routine initializes an IO_CSQ structure that describes a driver's cancel-safe IRP queue. Drivers can also use IoCsqInitializeEx to create an IRP queue with extended capabilities. For more information, see Cancel-

Safe IRP Queues.

Note that **IoCsq*Xxx*** routines use the **DriverContext**[3] member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system. Drivers that must also work for Windows 2000 and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IoCsqInitializeEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqInitializeEx** routine initializes the dispatch table for a cancel-safe IRP queue.

Syntax

```
NTSTATUS IoCsqInitializeEx(
  PIO_CSQ                      Csq,
  PIO_CSQ_INSERT_IRP_EX        CsqInsertIrp,
  PIO_CSQ_REMOVE_IRP           CsqRemoveIrp,
  PIO_CSQ_PEEK_NEXT_IRP        CsqPeekNextIrp,
  PIO_CSQ_ACQUIRE_LOCK         CsqAcquireLock,
  PIO_CSQ_RELEASE_LOCK         CsqReleaseLock,
  PIO_CSQ_COMPLETE_CANCELED_IRP CsqCompleteCanceledIrp
);
```

Parameters

`Csq`

Pointer to the IO_CSQ structure to be initialized by **IoCsqInitializeEx**.

`CsqInsertIrp`

Pointer to the driver-defined CsqInsertIrpEx function for the driver's cancel-safe IRP queue.

`CsqRemoveIrp`

Pointer to the driver-defined CsqRemoveIrp function for the driver's cancel-safe IRP queue.

`CsqPeekNextIrp`

Pointer to the driver-defined CsqPeekNextIrp function for the driver's cancel-safe IRP queue.

`CsqAcquireLock`

Pointer to the driver-defined CsqAcquireLock function for the driver's cancel-safe IRP queue.

`CsqReleaseLock`

Pointer to the driver-defined CsqReleaseLock function for the driver's cancel-safe IRP queue.

`CsqCompleteCanceledIrp`

Pointer to the driver-defined CsqCompleteCanceledIrp function for the driver's cancel-safe IRP queue.

Return Value

IoCsqInitializeEx returns STATUS_SUCCESS on success, or the appropriate error code on failure.

Remarks

The IoCsqInitialize and **IoCsqInitializeEx** routines initialize an IO_CSQ structure that describes a driver's cancel-safe IRP queue. You can use **IoCsqInitializeEx** to specify an IRP queue with extended capabilities instead of one

specified by **IoCsqInitialize**:

- The driver implements a CsqInsertIrpEx routine rather than a CsqInsertIrp routine. *CsqInsertIrpEx* takes an additional parameter, *InsertContext*. When the driver calls IoCsqInsertIrpEx to insert the IRP, it specifies the value that is passed as *InsertContext*.
- **IoCsqInsertIrpEx** returns the return value of *CsqInsertIrpEx*. Drivers can use the return value to indicate whether an insert operation succeeded or failed.

Otherwise, the effect of **IoCsqInitializeEx** is identical to that of **IoCsqInitialize**. For more information, see Cancel-Safe IRP Queues.

Note that **IoCsq*Xxx*** routines use the **DriverContext**[3] member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of the Windows operating system. The routine is also available in the Csq.lib library that ships with the Windows Driver Kit (WDK) and the Driver Development Kit (DDK) for Windows Server 2003. Drivers that must also work on Windows XP, Windows 2000, and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IO_CSQ_IRP_CONTEXT

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IoCsqInsertIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqInsertIrp** routine inserts an IRP in the driver's cancel-safe IRP queue.

Syntax

```
void IoCsqInsertIrp(
  PIO_CSQ             Csq,
  PIRP                Irp,
  PIO_CSQ_IRP_CONTEXT Context
);
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the driver's cancel-safe IRP queue. This structure must have been initialized by IoCsqInitialize or IoCsqInitializeEx.

`Irp`

Pointer to the IRP to be queued.

`Context`

Pointer to an IO_CSQ_IRP_CONTEXT structure. **IoCsqInsertIrp** initializes this structure with context information for the inserted IRP. The driver passes this value to IoCsqRemoveIrp to delete the IRP from the queue. *Context* can be **NULL** if the driver will not use **IoCsqRemoveIrp** to remove this IRP from the queue.

Return Value

None

Remarks

IoCsqInsertIrp uses the queue's dispatch routines to insert the IRP. The **IoCsqInsertIrp** routine:

1. Calls the queue's CsqAcquireLock routine to lock the queue.
2. Calls the queue's CsqInsertIrp routine to insert the IRP.
3. Marks the IRP as pending.
4. Calls the queue's CsqReleaseLock routine to unlock the queue.

If the IRP to be inserted has already been canceled, **IoCsqInsertIrp** does not attempt to insert the IRP into the queue.

Drivers can also use IoCsqInsertIrpEx to insert an IRP into the queue. For a queue that is specified by IoCsqInitializeEx, **IoCsqInsertIrpEx** provides additional capabilities. For more information, see Cancel-Safe IRP Queues.

Note that the **IoCsq*Xxx*** routines use the **DriverContext**[3] member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Callers of **IoCsqInsertIrp** must be running at an IRQL <= DISPATCH_LEVEL. The driver's callback routines must work correctly at that IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows. Drivers that must also work in Windows 2000 and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IoAllocateFree, IoReuseIrp, IrpCancelField, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IO_CSQ_IRP_CONTEXT

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IoCsqInsertIrpEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqInsertIrpEx** routine inserts an IRP into the driver's cancel-safe IRP queue.

Syntax

```
NTSTATUS IoCsqInsertIrpEx(
  PIO_CSQ             Csq,
  PIRP                Irp,
  PIO_CSQ_IRP_CONTEXT Context,
  PVOID               InsertContext
);
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the driver's cancel-safe IRP queue. This structure must have been initialized by IoCsqInitialize or IoCsqInitializeEx.

`Irp`

Pointer to the IRP to be queued.

`Context`

Pointer to an IO_CSQ_IRP_CONTEXT structure. **IoCsqInsertIrpEx** initializes this structure with context information for the inserted IRP. The driver passes this value to IoCsqRemoveIrp to delete the IRP from the queue. *Context* can be **NULL** if the driver will not use **IoCsqRemoveIrp** to remove this IRP from the queue.

`InsertContext`

Pointer to a driver-defined context value. This parameter is passed to the driver's CsqInsertIrpEx routine, if it has one. Otherwise, this parameter is ignored.

Return Value

If the *Csq* parameter was initialized with IoCsqInitialize, **IoCsqInsertIrpEx** always returns STATUS_SUCCESS. If *Csq* was initialized with **IoCsqInitializeEx**, **IoCsqInsertIrpEx** returns the value that was returned by the driver's CsqInsertIrpEx routine.

Remarks

IoCsqInsertIrpEx uses the queue's dispatch routines to insert the IRP. The **IoCsqInsertIrpEx** routine:

1. Calls the queue's CsqAcquireLock routine to lock the queue.
2. If the queue's IO_CSQ structure was initialized by IoCsqInitialize, **IoCsqInsertIrpEx** calls the queue's CsqInsertIrp routine to insert the IRP. If the queue's **IO_CSQ** structure was initialized by IoCsqInitializeEx, **IoCsqInsertIrpEx** calls the queue's CsqInsertIrpEx routine to insert the IRP, and passes the *InsertContext* parameter as the *InsertContext* parameter of *CsqInsertIrpEx*.

3. Calls the queue's CsqReleaseLock routine to unlock the queue.

If the IRP to be inserted has already been canceled, **IoCsqInsertIrpEx** does not attempt to insert the IRP into the queue.

For more information, see Cancel-Safe IRP Queues.

Note that **IoCsq*Xxx*** routines use the **DriverContext**[3] member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Callers of **IoCsqInsertIrpEx** must be running at an IRQL <= DISPATCH_LEVEL. The driver's callback routines must work correctly at this IRQL.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of the Windows operating system. The routine is also available in the Csq.lib library that ships with the Windows Driver Kit (WDK) and the Driver Development Kit (DDK) for Windows Server 2003. Drivers that must also work for on Windows XP, Windows 2000, and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IoAllocateFree, IoReuseIrp, IrpCancelField, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IO_CSQ_IRP_CONTEXT

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqRemoveIrp

IoCsqRemoveNextIrp

IoCsqRemoveIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqRemoveIrp** routine removes a particular IRP from the queue.

Syntax

```
PIRP IoCsqRemoveIrp(
  PIO_CSQ            Csq,
  PIO_CSQ_IRP_CONTEXT Context
);
```

Parameters

`Csq`

Pointer to the IO_CSQ structure for the driver's cancel-safe IRP queue. This structure must have been initialized by IoCsqInitialize or IoCsqInitializeEx.

`Context`

Pointer to the IO_CSQ_IRP_CONTEXT structure that identifies the IRP to remove. The **IO_CSQ_IRP_CONTEXT** structure is initialized by IoCsqInsertIrp or IoCsqInsertIrpEx when the IRP is first inserted in the queue.

Return Value

This routine returns a pointer to the IRP that was removed from the queue, or **NULL** if that IRP has been canceled.

Remarks

IoCsqRemoveIrp uses the queue's dispatch routines to remove the IRP. The **IoCsqRemoveIrp** routine:

1. Calls the queue's CsqAcquireLock routine to lock the queue.
2. Uses the *IrpContext* parameter to determine which IRP to remove, and calls the queue's CsqRemoveIrp routine to remove that IRP.
3. Calls the queue's CsqReleaseLock routine to unlock the queue.

Drivers can use the IoCsqRemoveNextIrp routine to remove an IRP that matches a specific criterion. For more information, see Cancel-Safe IRP Queues.

Note that **IoCsq*Xxx*** routines use the **DriverContext[**3**]** member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Callers of **IoCsqRemoveIrp** must be running at an IRQL <= DISPATCH_LEVEL. The driver's callback routines must work correctly at that IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows. Drivers that must also work in Windows 2000 and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IO_CSQ_IRP_CONTEXT

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveNextIrp

IoCsqRemoveNextIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoCsqRemoveNextIrp** routine removes the next matching IRP in the queue.

Syntax

```
PIRP IoCsqRemoveNextIrp(
  PIO_CSQ Csq,
  PVOID   PeekContext
);
```

Parameters

`Csq`

Pointer to the driver's dispatch table for cancel-safe IRP queues. The dispatch table must be initialized by IoCsqInitialize.

`PeekContext`

A pointer to a driver-defined context value. **IoCsqRemoveNextIrp** passes this parameter to the driver's CsqPeekNextIrp routine. For more information, see the following Remarks section.

Return Value

This routine returns a pointer to the next matching IRP in the queue, or **NULL** if no more IRPs are available. The routine only returns IRPs that have not yet been canceled.

Remarks

IoCsqRemoveNextIrp uses the queue's dispatch routines to remove the IRP. The **IoCsqRemoveNextIrp** routine:

1. Calls the queue's CsqAcquireLock routine to lock the queue.
2. Calls the queue's CsqPeekNextIrp routine to find the next matching IRP in the queue. **IoCsqRemoveNextIrp** passes the value of the *PeekContext* parameter as the *PeekContext* parameter of *CsqPeekNextIrp*. *CsqPeekNextIrp* returns a pointer to the next matching IRP, or **NULL** if there is no matching IRP.
3. If the return value of *CsqPeekNextIrp* is non-**NULL**, **IoCsqRemoveNextIrp** calls the queue's CsqRemoveIrp routine to remove the IRP.
4. Calls the queue's CsqReleaseLock routine to unlock the queue.

For more information, see Cancel-Safe IRP Queues.

Note that **IoCsq*Xxx*** routines use the **DriverContext**[3] member of the IRP to hold IRP context information. Drivers that use these routines to queue IRPs must leave that member unused.

Callers of **IoCsqRemoveNextIrp** must be running at an IRQL <= DISPATCH_LEVEL. The driver's callback routines must work correctly at that IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows. Drivers that must also work on Windows 2000 and Windows 98/Me can instead link to Csq.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

CsqAcquireLock

CsqCompleteCanceledIrp

CsqInsertIrp

CsqInsertIrpEx

CsqPeekNextIrp

CsqReleaseLock

CsqRemoveIrp

IO_CSQ

IoCsqInitialize

IoCsqInitializeEx

IoCsqInsertIrp

IoCsqInsertIrpEx

IoCsqRemoveIrp

IoDeleteDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDeleteDevice** routine removes a device object from the system, for example, when the underlying device is removed from the system.

Syntax

```
void IoDeleteDevice(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object to be deleted.

Return Value

None

Remarks

When handling a PnP IRP_MN_REMOVE_DEVICE request, a PnP driver calls **IoDeleteDevice** to delete any associated device objects. See Handling an IRP_MN_REMOVE_DEVICE Request for details.

A legacy driver should call this routine when it is being unloaded or when its DriverEntry routine encounters a fatal initialization error, such as being unable to properly initialize a physical device. This routine also is called when a driver reconfigures its devices dynamically. For example, a disk driver called to repartition a disk would call **IoDeleteDevice** to tear down the device objects representing partitions to be replaced.

A driver must release certain resources for which the driver supplied storage in its device extension before it calls **IoDeleteDevice**. For example, if the driver stores the pointer to its interrupt object(s) in the device extension, it must call IoDisconnectInterrupt before calling **IoDeleteDevice**.

A driver can call **IoDeleteDevice** only once for a given device object.

When a driver calls **IoDeleteDevice**, the I/O manager deletes the target device object if there are no outstanding references to it. However, if any outstanding references remain, the I/O manager marks the device object as "delete pending" and deletes the device object when the references are released.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	DeleteDevice, IrqlIoApcLte, PnpSurpriseRemove, RemoveLockCheck, HwStorPortProhibitedDDIs

See Also

IoCreateDevice

IoDisconnectInterrupt

IoDeleteSymbolicLink function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDeleteSymbolicLink** routine removes a symbolic link from the system.

Syntax

```
NTSTATUS IoDeleteSymbolicLink(
  PUNICODE_STRING SymbolicLinkName
);
```

Parameters

`SymbolicLinkName`

Pointer to a buffered Unicode string that is the user-visible name for the symbolic link.

Return Value

IoDeleteSymbolicLink returns STATUS_SUCCESS if the symbolic link object is deleted.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoCreateSymbolicLink

IoCreateUnprotectedSymbolicLink

IoDeassignArcName

IoDetachDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDetachDevice** routine releases an attachment between the caller's device object and a lower driver's device object.

Syntax

```
void IoDetachDevice(
  PDEVICE_OBJECT TargetDevice
);
```

Parameters

`TargetDevice`

Pointer to the lower driver's device object. The caller previously called IoAttachDevice or IoAttachDeviceToDeviceStack successfully to get this pointer.

Return Value

None

Remarks

IoDetachDevice decrements the reference count of the *TargetDevice* object. If the reference count goes to zero and the lower driver has been marked for an unload operation, the lower driver is unloaded.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive3, PnpSurpriseRemove, PowerIrpDDis, RemoveLockCheck, HwStorPortProhibitedDDIs

See Also

IoAttachDevice

IoAttachDeviceToDeviceStack

IoDisconnectInterrupt function

1/25/2019 • 2 minutes to read • Edit Online

The **IoDisconnectInterrupt** routine releases a device driver's set of interrupt object(s) when the device is paused or removed, or when the driver is being unloaded.

Syntax

```
void IoDisconnectInterrupt(
  PKINTERRUPT InterruptObject
);
```

Parameters

`InterruptObject`

A pointer to a KINTERRUPT structure. The caller obtained this pointer from the IoConnectInterrupt call that previously connected the interrupt or interrupts.

Return Value

None.

Remarks

The driver should configure the device to issue interrupts only when these interrupts are connected. Failure to prevent a device from issuing interrupts when the interrupts are disconnected might cause system instability. For example, if a device shares a level-triggered interrupt line with other devices, and the device issues an interrupt request when the device's interrupts are disconnected, the other devices on the line will not acknowledge the interrupt and the interrupt will continue to fire. Before calling **IoDisconnectInterrupt**, the driver should configure the device to stop issuing interrupts. After calling IoConnectInterrupt, the driver should configure the device to start issuing interrupts.

If the driver stored the pointer to its interrupt object(s) in the device extension of its device object or in the controller extension of its controller object, it must call **IoDisconnectInterrupt** before it calls IoDeleteDevice or IoDeleteController.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive4, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoConnectInterrupt

IoDeleteController

IoDeleteDevice

KINTERRUPT

IoDisconnectInterruptEx function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibIoDisconnectInterruptEx function.

```
#define IoDisconnectInterruptEx WdmlibIoDisconnectInterruptEx
```

Syntax

```
void IoDisconnectInterruptEx(
  PIO_DISCONNECT_INTERRUPT_PARAMETERS Parameters
);
```

Parameters

`Parameters`

For more information, see the WdmlibIoDisconnectInterruptEx function.

Return Value

For more information, see the WdmlibIoDisconnectInterruptEx function.

Requirements

Minimum supported client	Available starting with Windows Vista. Drivers that must also work Microsoft Windows 2000, Windows XP, or Windows Server 2003 can instead link to Iointex.lib to use the routine.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib; Iointex.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

WdmlibIoDisconnectInterruptEx

IoForwardIrpSynchronously function

1/25/2019 • 2 minutes to read • Edit Online

The **IoForwardIrpSynchronously** routine sends an IRP to a specified driver and waits for that driver to complete the IRP.

Syntax

```
BOOLEAN IoForwardIrpSynchronously(
  PDEVICE_OBJECT DeviceObject,
  PIRP           Irp
);
```

Parameters

`DeviceObject`

Specifies the device object of the driver to which the IRP is sent.

`Irp`

Specifies the IRP to send.

Return Value

IoForwardIrpSynchronously returns **TRUE** if the IRP is sent successfully. Otherwise, it returns **FALSE**.

Remarks

Use the **IoForwardIrpSynchronously** routine to synchronously forward the current IRP to the next driver in the stack. (To asynchronously forward the IRP, use the IoCopyCurrentIrpStackLocationToNext, IoCallDriver, and IoSetCompletionRoutine routines.)

The **IoForwardIrpSynchronously** routine copies the current stack location to the next stack location. That stack location is then used as the current stack location by the driver specified in *DeviceObject*. If no next stack location is available, the routine returns **FALSE**. If the routine returns **TRUE**, then the IRP has been sent to the specified driver, and that driver has completed its processing of this IRP.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	PnpIrpCompletion

See Also

IoCallDriver

IoCopyCurrentIrpStackLocationToNext

IoSetCompletionRoutine

IoFreeErrorLogEntry function

1/25/2019 • 2 minutes to read • Edit Online

The **IoFreeErrorLogEntry** routine frees an unused error log entry.

Syntax

```
void IoFreeErrorLogEntry(
  PVOID ElEntry
);
```

Parameters

`ElEntry`

Pointer to an error log packet allocated by IoAllocateErrorLogEntry.

Return Value

None

Remarks

Drivers use **IoFreeErrorLogEntry** to free an error log entry allocated by IoAllocateErrorLogEntry. IoWriteErrorLogEntry also frees any error log entries passed to it, so drivers must not call both on the same log entry.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

See Also

IO_ERROR_LOG_PACKET

IoAllocateErrorLogEntry

IoWriteErrorLogEntry

IoFreeIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoFreeIrp** routine releases a caller-allocated IRP from the caller's IoCompletion routine.

Syntax

```
void IoFreeIrp(
  PIRP Irp
);
```

Parameters

`Irp`

Pointer to the IRP that is to be released.

Return Value

None

Remarks

This routine is the reciprocal to IoAllocateIrp or IoBuildAsynchronousFsdRequest. The released IRP must have been allocated by the caller.

This routine also releases an IRP allocated with IoMakeAssociatedIrp in which the caller set up its IoCompletion routine that returns STATUS_MORE_PROCESSING_REQUIRED for the associated IRP.

IoFreeIrp does not free any MDLs that might be attached to the IRP. The driver that frees the IRP must explicitly free these MDLs. In addition, if the physical pages that are described by an MDL are locked, the driver must unlock the pages before it frees the MDL. However, the driver does not need to explicitly unmap these pages. Instead, IoFreeMdl automatically unmaps the pages when it frees the MDL. For a code example that shows how to free an MDL chain, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IoAllocateFree, IoBuildDeviceControlNoFree, IoBuildFsdFree, IoBuildSynchronousFsdRequestNoFree, HwStorPortProhibitedDDIs, IoFreeIrp

See Also

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoCompletion

IoMakeAssociatedIrp

IoSetCompletionRoutine

IoFreeMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **IoFreeMdl** routine releases a caller-allocated memory descriptor list (MDL).

Syntax

```
void IoFreeMdl(
  PMDL Mdl
);
```

Parameters

`Mdl`

Pointer to the MDL to be released.

Return Value

None

Remarks

If a driver allocates an MDL to describe a buffer, it must explicitly release the MDL when operations on the buffer are done.

If the MDL is a partial MDL, **IoFreeMdl** unmaps any pages that have been mapped to the MDL.

If the physical pages that are described by the MDL are locked, the driver must unlock the pages before it frees the MDL.

This routine frees only the specified MDL. Any MDLs that are chained to this MDL must be explicitly freed through additional calls to this routine. For a code example that shows how to free an MDL chain, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoAllocateMdl

IoBuildPartialMdl

IoFreeWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoFreeWorkItem** routine frees a work item that was allocated by IoAllocateWorkItem.

Syntax

```
void IoFreeWorkItem(
  PIO_WORKITEM IoWorkItem
);
```

Parameters

`IoWorkItem`

Pointer to an IO_WORKITEM structure that was returned by a previous call to IoAllocateWorkItem.

Return Value

None

Remarks

Only free a work item that is not currently queued. The system dequeues a work item before it runs the work item's callback routine, so **IoFreeWorkItem** can be called from within the WorkItem or WorkItemEx routine for the work item.

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_WORKITEM

IoGetAffinityInterrupt function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibIoGetAffinityInterrupt function.

```
#define IoGetAffinityInterrupt WdmlibIoGetAffinityInterrupt
```

Syntax

```
NTSTATUS IoGetAffinityInterrupt(
  PKINTERRUPT     InterruptObject,
  PGROUP_AFFINITY GroupAffinity
);
```

Parameters

`InterruptObject`

For more information, see the WdmlibIoGetAffinityInterrupt function.

`GroupAffinity`

For more information, see the WdmlibIoGetAffinityInterrupt function.

Return Value

For more information, see the WdmlibIoGetAffinityInterrupt function.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

WdmlibIoGetAffinityInterrupt

IoGetAttachedDeviceReference function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetAttachedDeviceReference** routine returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.

Syntax

```
PDEVICE_OBJECT IoGetAttachedDeviceReference(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the topmost attached device object is retrieved.

Return Value

IoGetAttachedDeviceReference returns a pointer to the highest level device object in a stack of attached device objects after incrementing the reference count on the object.

Remarks

If the device object at *DeviceObject* has no device objects attached to it, *DeviceObject* and the returned pointer are equal.

Device driver writers must ensure that when they have completed all operations that required them to make this call, that they call ObDereferenceObject with the device object pointer returned by this routine. Failure to do so will prevent the system from freeing or deleting the device object because of an outstanding reference count.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	DanglingDeviceObjectReference, HwStorPortProhibitedDDIs

See Also

ObDereferenceObject

IoGetBootDiskInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetBootDiskInformation** routine returns information describing the boot and system disks.

Syntax

```
NTSTATUS IoGetBootDiskInformation(
  PBOOTDISK_INFORMATION BootDiskInformation,
  ULONG                 Size
);
```

Parameters

`BootDiskInformation`

Pointer to a caller-allocated buffer that the routine uses to return information about the boot and system disks. The routine fills this buffer in with either a BOOTDISK_INFORMATION or a BOOTDISK_INFORMATION_EX structure.

`Size`

Specifies the size, in bytes, of the buffer specified by *BootDiskInformation*. Should be either **sizeof**(**BOOTDISK_INFORMATION**) or **sizeof**(**BOOTDISK_INFORMATION_EX**).

Return Value

IoGetBootDiskInformation returns one of the following status values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully returned the requested information in the buffer specified by *BootDiskInformation*.
STATUS_INVALID_PARAMETER	The value of *Size* is less than the size, in bytes, of a **BOOTDISK_INFORMATION** structure.

STATUS_TOO_LATE	The driver called the routine after the system has already booted. Only boot and system drivers can call **IoGetBootDiskInformation**, and then only in their DriverEntry or AddDevice routines.

Remarks

IoGetBootDiskInformation can be called only by a boot driver. This driver should call **IoGetBootDiskInformation** in a Reinitialize callback routine that the driver registers by calling the IoRegisterBootDriverReinitialization routine.

On Windows XP and later versions of Windows, if the *Size* parameter is **sizeof**(**BOOTDISK_INFORMATION_EX**) or larger, the routine returns a BOOTDISK_INFORMATION_EX structure in the *BootDiskInformation* buffer. Otherwise, if *Size* is at least **sizeof**(**BOOTDISK_INFORMATION**), the routine returns a BOOTDISK_INFORMATION structure.

On Windows 2000, the routine returns only the **BOOTDISK_INFORMATION** structure.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

BOOTDISK_INFORMATION

BOOTDISK_INFORMATION_EX

IoRegisterBootDriverReinitialization

Reinitialize

IoGetContainerInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetContainerInformation** routine provides information about the current state of a user session.

Syntax

```
NTSTATUS IoGetContainerInformation(
  IO_CONTAINER_INFORMATION_CLASS InformationClass,
  PVOID                          ContainerObject,
  PVOID                          Buffer,
  ULONG                          BufferLength
);
```

Parameters

`InformationClass`

Specifies the class of events for which the caller (driver) requests information. Set this parameter to the following IO_CONTAINER_INFORMATION_CLASS enumeration value:

- **IoSessionStateInformation**

For more information, see the following Remarks section.

`ContainerObject`

A pointer to an opaque, system object supplied by the I/O manager. For *InformationClass* = **IoSessionStateInformation**, set this parameter to the *SessionObject* parameter value that is provided by the I/O manager during the call to the driver's IO_SESSION_NOTIFICATION_FUNCTION function.

`Buffer`

A pointer to a caller-allocated buffer into which this routine writes the state information for the event class specified by *InformationClass*. For *InformationClass* = **IoSessionStateInformation**, the routine writes an IO_SESSION_STATE_INFORMATION structure to the buffer. The buffer must be large enough to contain this structure.

`BufferLength`

The size, in bytes, of the buffer pointed to by *Buffer*. For *InformationClass* = IoSessionStateInformation, *BufferLength* must be at least **sizeof**(**IO_SESSION_STATE_INFORMATION**).

Return Value

IoGetContainerInformation returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION

STA TUS _IN VAL ID_ PAR AM ETE R_1	Parameter *InformationClass* is not a valid IO_CONTAINER_INFORMATION_CLASS enumeration constant.
STA TUS _IN VAL ID_ PAR AM ETE R_2	Parameter *ContainerObject* is **NULL**.
STA TUS _IN VAL ID_ PAR AM ETE R_4	Parameter *BufferLength* is too small for the information class specified by *InformationClass*.

Remarks

This routine can potentially support queries for a variety of information classes. In Windows 7, this routine supports only queries for **IoSessionStateInformation** information, which is status information about user sessions.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IO_CONTAINER_INFORMATION_CLASS

IO_CONTAINER_NOTIFICATION_CLASS

IO_SESSION_STATE_INFORMATION

IoGetCurrentIrpStackLocation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetCurrentIrpStackLocation** routine returns a pointer to the caller's I/O stack location in the specified IRP.

Syntax

```
__drv_aliasesMem PIO_STACK_LOCATION IoGetCurrentIrpStackLocation(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to the IRP.

Return Value

IoGetCurrentIrpStackLocation returns a pointer to an IO_STACK_LOCATION structure that contains the I/O stack location for the driver.

Remarks

Every driver must call **IoGetCurrentIrpStackLocation** with each IRP it is sent in order to get any parameters for the current request. Unless a driver supplies a dispatch routine for each **IRP_MJ_*XXX*** code that the driver handles, the driver also must check its I/O stack location in the IRP to determine what operation is being requested.

If a driver is passing the same parameters that it received to the next-lower driver, it should call IoCopyCurrentIrpStackLocationToNext or IoSkipCurrentIrpStackLocation instead of getting a pointer to the next-lower stack location and copying the parameters manually.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IO_STACK_LOCATION

IRP

IoCallDriver

IoGetNextIrpStackLocation

IoSetNextIrpStackLocation

IoGetCurrentProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetCurrentProcess** routine returns a pointer to the current process. Call **PsGetCurrentProcess** instead of **IoGetCurrentProcess**.

Syntax

```
PEPROCESS IoGetCurrentProcess(

);
```

Parameters

This function has no parameters.

Return Value

IoGetCurrentProcess returns a pointer to the current process.

Remarks

This routine is identical to **PsGetCurrentProcess**.

```
#define PsGetCurrentProcess IoGetCurrentProcess
```

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

PsGetCurrentThread

IoGetDeviceInterfaceAlias function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDeviceInterfaceAlias** routine returns the alias device interface of the specified device interface instance, if the alias exists.

Syntax

```
NTSTATUS IoGetDeviceInterfaceAlias(
  PUNICODE_STRING SymbolicLinkName,
  const GUID      *AliasInterfaceClassGuid,
  PUNICODE_STRING AliasSymbolicLinkName
);
```

Parameters

`SymbolicLinkName`

Pointer to the name of the device interface instance for which to retrieve an alias. The caller typically received this string from a call to IoGetDeviceInterfaces or in a PnP notification structure.

`AliasInterfaceClassGuid`

Pointer to a GUID specifying the interface class of the alias to retrieve.

`AliasSymbolicLinkName`

Specifies a pointer to a **NULL** Unicode string. On successful return, *AliasSymbolicLinkName*.**Buffer** points to a string containing the name of the alias. The caller must free the Unicode string with RtlFreeUnicodeString when it is no longer needed.

Return Value

IoGetDeviceInterfaceAlias returns STATUS_SUCCESS if the call was successful. Possible error return values are described following.

RETURN CODE	DESCRIPTION
STATUS_OBJECT_NAME_NOT_FOUND	Possibly indicates that there is no alias of the specified interface class.

STA TUS _OB JEC T_P AT H_N OT_ FOU ND	Possibly indicates that there is no alias of the specified interface class.
STA TUS _IN VAL ID_ HA NDL E	Possibly indicates an invalid *SymbolicLinkName* or an invalid *AliasClassGuid*.

Remarks

Device interfaces are considered aliases if they are exposed by the same underlying device and have identical interface reference strings, but are of different interface classes.

The *SymbolicLinkName* parameter specifies a device interface instance for a particular device, belonging to a particular interface class, with a particular reference string. **IoGetDeviceInterfaceAlias** returns another device interface instance for the same device and reference string, but of a different interface class, if it exists.

For example, the function driver for a fault-tolerant volume could register and set two device interfaces, one of the fault-tolerant-volume interface class and one of the volume interface class. Another driver could call **IoGetDeviceInterfaceAlias** with the symbolic link for one of the interfaces and ask whether the other interface exists by specifying its interface class.

Two device interfaces with **NULL** reference strings are aliases if they are exposed by the same underlying device and have different interface class GUIDs.

Callers of **IoGetDeviceInterfaceAlias** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoRegisterDeviceInterface

RtlFreeUnicodeString

IoGetDeviceInterfacePropertyData function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDeviceInterfacePropertyData** routine retrieves the current value of a device interface property.

Syntax

```
NTSTATUS IoGetDeviceInterfacePropertyData(
  PUNICODE_STRING  SymbolicLinkName,
  const DEVPROPKEY *PropertyKey,
  LCID             Lcid,
  ULONG            Flags,
  ULONG            Size,
  PVOID            Data,
  PULONG           RequiredSize,
  PDEVPROPTYPE     Type
);
```

Parameters

`SymbolicLinkName`

A pointer to a string that identifies the device interface instance. This string was obtained from a previous call to the IoGetDeviceInterfaces, IoGetDeviceInterfaceAlias, or IoRegisterDeviceInterface routine.

`PropertyKey`

A pointer to a DEVPROPKEY structure that contains the device interface property key.

`Lcid`

Specifies a locale identifier. Set this parameter either to a language-specific LCID value or to **LOCALE_NEUTRAL**. The **LOCALE_NEUTRAL** LCID specifies that the property is language-neutral (that is, not specific to any language). Do not set this parameter to **LOCALE_SYSTEM_DEFAULT** or **LOCALE_USER_DEFAULT**. For more information about language-specific LCID values, see LCID Structure.

`Flags`

Reserved for system use. Drivers should set this value to zero.

`Size`

Specifies the size, in bytes, of the buffer that *Data* points to.

`Data`

A pointer to a caller-allocated buffer into which the routine writes the device interface property data.

`RequiredSize`

A pointer to a ULONG variable into which **IoGetDeviceInterfacePropertyData** writes the required size of the property data. If the routine succeeds, the required size value is the number of bytes that the routine writes to the output buffer that *Data* points to. If the routine returns STATUS_BUFFER_TOO_SMALL, the required size value is the size of the buffer that the caller should allocate for this property value.

Type

A pointer to a DEVPROPTYPE variable. If **IoGetDeviceInterfacePropertyData** successfully retrieves the property data, the routine writes the property type value to this variable. This value indicates the type of property data that is in the *Data* buffer.

Return Value

IoGetDeviceInterfacePropertyData returns STATUS_SUCCESS if it is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The buffer that *Data* points to is too small to contain the property data. **RequiredSize* contains the required buffer length.
STATUS_UNSUCCESSFUL	The specified LCID value is not valid.
STATUS_NOT_IMPLEMENTED	The specified property is not supported.

Remarks

Kernel-mode drivers use the **IoGetDeviceInterfacePropertyData** routine to retrieve device interface properties that are defined as part of the unified device property model. For more information about device interface properties, see Device Properties.

Drivers can use the IoSetDeviceInterfacePropertyData routine to modify a device interface property.

Callers of **IoGetDeviceInterfacePropertyData** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available in Windows 8 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

DEVPROPKEY

DEVPROPTYPE

IoSetDeviceInterfacePropertyData

IoGetDeviceInterfaces function

1/25/2019 • 3 minutes to read • Edit Online

The **IoGetDeviceInterfaces** routine returns a list of device interface instances of a particular device interface class (such as all devices on the system that support a HID interface).

Syntax

```
NTSTATUS IoGetDeviceInterfaces(
  const GUID     *InterfaceClassGuid,
  PDEVICE_OBJECT PhysicalDeviceObject,
  ULONG          Flags,
  PZZWSTR        *SymbolicLinkList
);
```

Parameters

`InterfaceClassGuid`

Pointer to a class GUID specifying the device interface class. The GUIDs for a class should be in a device-specific header file.

`PhysicalDeviceObject`

Pointer to an optional PDO that narrows the search to only the device interface instances of the device represented by the PDO.

`Flags`

Specifies flags that modify the search for device interfaces. Only one flag is currently defined, and is described in the following table.

FLAG	MEANING
DEVICE_INTERFACE_INCLUDE_NONACTIVE	Return disabled device interface instances in addition to enabled interface instances.

When searching for a device that supports a particular interface class, the caller requires an enabled interface instance and thus does not set the DEVICE_INTERFACE_INCLUDE_NONACTIVE flag.

A driver typically sets the DEVICE_INTERFACE_INCLUDE_NONACTIVE flag to locate disabled interface instances that the driver must enable. For example, the class installer for the device might have been directed by the INF file to register one or more interface instances for the device. The interface instances would be registered but are not usable until they are enabled by the driver (using IoSetDeviceInterfaceState). To narrow the list of interface instances returned to only those exposed by a given device, a driver can specify a *PhysicalDeviceObject*.

`SymbolicLinkList`

A pointer to a wide character pointer to which the routine, if successful, writes the base address of a buffer that contains a list of Unicode strings. These strings are symbolic link names that identify the device interface instances that match the search criteria. Each Unicode string in the list is null-terminated; the end of the whole list is marked by an additional null character. The routine allocates the buffer for these strings from paged system memory. The caller is responsible for freeing the buffer (by calling the ExFreePool routine) when it is no longer needed.

If no device interface instances match the search criteria, this routine returns STATUS_SUCCESS and the string contains a single NULL character.

Return Value

IoGetDeviceInterfaces returns STATUS_SUCCESS if the call was successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that *PhysicalDeviceObject* was not a valid PDO pointer.

Remarks

IoGetDeviceInterfaces returns a list of device interface instances that match the search criteria. A kernel-mode component typically calls this routine to get a list of all enabled device interface instances of a particular device interface class. Such a component can get a pointer to the file object and/or the device object for an interface by calling the IoGetDeviceObjectPointer or ZwCreateFile routine. The device object pointer returned by **IoGetDeviceObjectPointer** points to the top of the device stack for the device and can be used in calls to the IoCallDriver routine.

If there is a default interface for the requested device interface class, it is listed first in *SymbolicLinkList*. Default interfaces can be set by user mode, but not by kernel mode.

The format of a symbolic link name is opaque; the caller should not attempt to parse a symbolic link name.

Symbolic links for device interface instances can be used across system boots.

To be notified when additional device interface instances of a particular class are enabled on the system, register for notification of a device class change by calling the IoRegisterPlugPlayNotification routine.

Callers of **IoGetDeviceInterfaces** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ExFreePool

IoCallDriver

IoGetDeviceObjectPointer

IoRegisterDeviceInterface

IoRegisterPlugPlayNotification

IoSetDeviceInterfaceState

ZwCreateFile

IoGetDeviceNumaNode function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDeviceNumaNode** routine gets the node number of a device.

Syntax

```
NTSTATUS IoGetDeviceNumaNode(
  PDEVICE_OBJECT Pdo,
  PUSHORT        NodeNumber
);
```

Parameters

`Pdo`

A pointer to a physical device object (PDO). This parameter points to a DEVICE_OBJECT structure that represents a physical device.

`NodeNumber`

A pointer to a location into which the routine writes the node number, if the node number is known.

Return Value

IoGetDeviceNumaNode returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_NO_FOUND	The node number of this device is unknown.
STATUS_INVALID_PARAMETER	The *Pdo* parameter is **NULL** or does not point to a valid device object.

Remarks

In a non-uniform memory access (NUMA) multiprocessor architecture, a node is a collection of processors that

share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

A device (for example, a network controller or storage controller) that is connected to a particular node can access the memory in this node faster than it can access the memory in other nodes. The *Pdo* parameter points to a PDO that represents the bus connection between the device and the node.

If a NUMA multiprocessor system contains *n* nodes, the nodes are numbered from 0 to *n*-1. To obtain the highest node number, call the KeQueryHighestNodeNumber routine.

After the system is initialized, the node count remains fixed while the system continues to run. This count might include memory-only nodes, which are nodes that contain memory but no active logical processors. A node in which all the logical processors are inactive is effectively a memory-only node until the first processor in the node starts to run. Do not assume that the node that a device is connected to contains active processors.

If a system does not have a NUMA architecture, the routine writes zero to the location that the *NodeNumber* parameter points to. An example of non-NUMA architecture is a symmetric multiprocessor system (SMP).

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating systems.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DEVICE_OBJECT

KeQueryHighestNodeNumber

IoGetDeviceObjectPointer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDeviceObjectPointer** routine returns a pointer to the top object in the named device object's stack and a pointer to the corresponding file object, if the requested access to the objects can be granted.

Syntax

```
NTSTATUS IoGetDeviceObjectPointer(
  PUNICODE_STRING ObjectName,
  ACCESS_MASK     DesiredAccess,
  PFILE_OBJECT    *FileObject,
  PDEVICE_OBJECT  *DeviceObject
);
```

Parameters

`ObjectName`

Pointer to a buffer that contains a Unicode string that is the name of the device object.

`DesiredAccess`

Specifies the ACCESS_MASK value that represents the desired access. Usually *DesiredAccess* is FILE_READ_DATA. Infrequently, the FILE_WRITE_DATA, or FILE_ALL_ACCESS access rights are specified.

`FileObject`

Pointer to the file object that represents the corresponding device object to user-mode code if the call is successful.

`DeviceObject`

Pointer to the device object that represents the named logical, virtual, or physical device if the call is successful.

Return Value

IoGetDeviceObjectPointer returns STATUS_SUCCESS if it is successful. Possible error return values include the following status codes:

STATUS_OBJECT_TYPE_MISMATCH

STATUS_INVALID_PARAMETER

STATUS_PRIVILEGE_NOT_HELD

STATUS_INSUFFICIENT_RESOURCES

STATUS_OBJECT_NAME_INVALID

Remarks

IoGetDeviceObjectPointer establishes a "connection" between the caller and the next-lower-level driver. A successful caller can use the returned device object pointer to initialize its own device objects. It can also be used as an argument to IoAttachDeviceToDeviceStack, IoCallDriver, and any routine that creates IRPs for lower drivers.

The returned pointer is a required argument to **IoCallDriver**.

This routine also returns a pointer to the corresponding file object. When unloading, a driver can dereference the file object as a means of indirectly dereferencing the device object. To do so, the driver calls ObDereferenceObject from its Unload routine, passing the file object pointer returned by **IoGetDeviceObjectPointer**. Failure to dereference the device object in a driver's Unload routine prevents the next-lower driver from being unloaded. However, drivers that close the file object before the unload process must take out an extra reference on the device object before dereferencing the file object. Otherwise, dereferencing the file object can lead to a premature deletion of the device object.

To get a pointer to the highest-level driver in the file system driver stack, a driver must ensure that the file system is mounted; if it is not, this routine traverses the storage device stack. To ensure that the file system is mounted on the storage device, the driver must specify an appropriate access mask, such as FILE_READ_DATA or FILE_WRITE_ATTRIBUTES, in the *DesiredAccess* parameter. Specifying FILE_READ_ATTRIBUTES does not cause the file system to be mounted.

After any higher-level driver has chained itself over another driver by successfully calling this routine, the higher-level driver must set the **StackSize** field in its device object to that of the next-lower-level driver's device object plus one.

Callers of **IoGetDeviceObjectPointer** must be running at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

DEVICE_OBJECT

IoAllocateIrp

IoAttachDevice

IoAttachDeviceToDeviceStack

IoCallDriver

ObDereferenceObject

ObReferenceObjectByPointer

IoGetDeviceProperty function

1/25/2019 • 4 minutes to read • Edit Online

The **IoGetDeviceProperty** routine retrieves information about a device such as configuration information and the name of its PDO.

Syntax

```
NTSTATUS IoGetDeviceProperty(
  PDEVICE_OBJECT           DeviceObject,
  DEVICE_REGISTRY_PROPERTY DeviceProperty,
  ULONG                    BufferLength,
  PVOID                    PropertyBuffer,
  PULONG                   ResultLength
);
```

Parameters

`DeviceObject`

Pointer to the physical device object (PDO) for the device being queried.

`DeviceProperty`

Specifies the device property being requested. Must be one of the following DEVICE_REGISTRY_PROPERTY enumeration values:

DevicePropertyAddress

Requests the address of the device on the bus. *PropertyBuffer* points to a ULONG.

The interpretation of this address is bus-specific. The caller of this routine should call the routine again to request the **DevicePropertyBusTypeGuid**, or possibly the **DevicePropertyLegacyBusType**, so it can interpret the address. An address value of 0xFFFFFFFF indicates that the underlying bus driver did not supply a bus address for the device.

DevicePropertyBootConfiguration

Requests the hardware resources assigned to the device by the firmware, in raw form. *PropertyBuffer* points to a CM_RESOURCE_LIST structure.

DevicePropertyBootConfigurationTranslated

The hardware resources assigned to the device by the firmware, in translated form. *PropertyBuffer* points to a **CM_RESOURCE_LIST** structure.

DevicePropertyBusNumber

Requests the legacy bus number of the bus the device is connected to. *PropertyBuffer* points to a ULONG.

DevicePropertyBusTypeGuid

Requests the GUID for the bus that the device is connected to. The system-defined bus type GUIDs are listed in the Wdmguid.h header file. *PropertyBuffer* points to a GUID, which is a 16-byte structure that contains the GUID in binary form.

DevicePropertyClassGuid

Requests the GUID for the device's setup class.*PropertyBuffer* points to a NULL-terminated array of WCHAR. This routine returns the GUID in a string format as follows, where each "c" represents a hexadecimal character:

{cccccccc-cccc-cccc-cccc-cccccccccccc}

DevicePropertyClassName

Requests the name of the device's setup class, in text format. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyCompatibleIDs

Requests the compatible IDs reported by the device. *PropertyBuffer* points to a REG_MULTI_SZ value.

DevicePropertyDeviceDescription

Requests a string describing the device, such as "Microsoft PS/2 Port Mouse", typically defined by the manufacturer. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyDriverKeyName

Requests the name of the driver-specific registry key. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyEnumeratorName

Requests the name of the enumerator for the device, such as "PCI" or "root". *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyFriendlyName

Requests a string that can be used to distinguish between two similar devices, typically defined by the class installer. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyHardwareID

Requests the hardware IDs provided by the device that identify the device. *PropertyBuffer* points to a REG_MULTI_SZ value.

DevicePropertyInstallState

Requests the device's installation state. The installation state is returned as a DEVICE_INSTALL_STATE enumeration value.

DevicePropertyLegacyBusType

Requests the bus type, such as PCIBus or PCMCIABus. *PropertyBuffer* points to an INTERFACE_TYPE enumeration value.

DevicePropertyLocationInformation

Requests information about the device's location on the bus; the interpretation of this information is bus-specific. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyManufacturer

Requests a string identifying the manufacturer of the device. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyPhysicalDeviceObjectName

Requests the name of the PDO for this device. *PropertyBuffer* points to a NULL-terminated WCHAR string.

DevicePropertyRemovalPolicy

Requests the device's current removal policy. The operating system uses this value as a hint to determine how the device is normally removed. The *PropertyBuffer* parameter points to a DEVICE_REMOVAL_POLICY enumeration value.

DevicePropertyUINumber

Requests a number associated with the device that can be displayed in the user interface. *PropertyBuffer* points to a ULONG value.

This number is typically a user-perceived slot number, such as a number printed next to the slot on the board, or some other number that makes locating the physical device easier for the user. If the device is on a bus that has no UI number convention, or if the bus driver for the device cannot determine the UI number, this value is 0xFFFFFFFF.

`BufferLength`

Specifies the size, in bytes, of the caller-supplied *PropertyBuffer*.

`PropertyBuffer`

Pointer to a caller-supplied buffer to receive the property information. The buffer can be allocated from pageable memory. The type of the buffer is determined by the *DeviceProperty* (see above).

`ResultLength`

Pointer to a ULONG to receive the size of the property information returned at *PropertyBuffer*. If **IoGetDeviceProperty** returns STATUS_BUFFER_TOO_SMALL, it sets this parameter to the required buffer length.

Return Value

IoGetDeviceProperty returns STATUS_SUCCESS if the call was successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The buffer at *PropertyBuffer* was too small. *ResultLength* points to the required buffer length.
STATUS_INVALID_PARAMETER_2	The given *DeviceProperty* is not one of the properties handled by this routine.
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that the given *DeviceObject* was not a valid PDO pointer.

Remarks

IoGetDeviceProperty retrieves device setup information from the registry. Use this routine, rather than accessing the registry directly, to insulate a driver from differences across platforms and from possible changes in the registry structure.

For many *DeviceProperty* requests, it can take two or more calls to **IoGetDeviceProperty** to determine the required *BufferLength*. The first call should use a best-guess value. If the return status is STATUS_BUFFER_TOO_SMALL, the driver should free its current buffer, allocate a buffer of the size returned in *ResultLength*, and call **IoGetDeviceProperty** again. Because some of the setup properties are dynamic, the data size can change between the time the required size is returned and driver calls this routine again. Therefore, drivers should call **IoGetDeviceProperty** inside a loop that runs until the return status is not STATUS_BUFFER_TOO_SMALL.

Function drivers that support devices on a legacy bus and a PnP bus can use the **DevicePropertyBusNumber**, **DevicePropertyBusTypeGuid**, and **DevicePropertyLegacyBusType** properties to distinguish between the buses.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

CM_RESOURCE_LIST

DEVICE_REMOVAL_POLICY

ExAllocatePoolWithTag

GUID

INTERFACE_TYPE

IO_RESOURCE_REQUIREMENTS_LIST

IoGetDevicePropertyData function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDevicePropertyData** routine retrieves the current setting for a device property.

Syntax

```
NTSTATUS IoGetDevicePropertyData(
  PDEVICE_OBJECT   Pdo,
  const DEVPROPKEY *PropertyKey,
  LCID             Lcid,
  ULONG            Flags,
  ULONG            Size,
  PVOID            Data,
  PULONG           RequiredSize,
  PDEVPROPTYPE     Type
);
```

Parameters

`Pdo`

A pointer to the physical device object (PDO) for the device that is being queried.

`PropertyKey`

A pointer to a DEVPROPKEY structure that specifies the device property key.

`Lcid`

A locale identifier. Set this parameter either to a language-specific LCID value or to **LOCALE_NEUTRAL**. The **LOCALE_NEUTRAL** LCID specifies that the property is language-neutral (that is, not specific to any language). Do not set this parameter to **LOCALE_SYSTEM_DEFAULT** or **LOCALE_USER_DEFAULT**. For more information about language-specific LCID values, see LCID Structure.

`Flags`

Reserved for system use. Drivers should set this value to 0.

`Size`

The size, in bytes, of the buffer that *Data* points to.

`Data`

A pointer to the device property data.

`RequiredSize`

A pointer to a ULONG to receive the size of the property information that is returned at *Data*. If **IoGetDevicePropertyData** returns STATUS_BUFFER_TOO_SMALL, the caller can use this value to allocate a buffer of the correct size.

`Type`

A pointer to a DEVPROPTYPE value. If **IoGetDevicePropertyData** completes successfully, the routine uses *Type*

to supply the type of data that is returned in the *Data* buffer.

Return Value

IoGetDevicePropertyData returns an NTSTATUS value. This routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The *Data* buffer contains the retrieved data. **Type* contains the type of the retrieved data.
STATUS_BUFFER_TOO_SMALL	The *Data* buffer is too small. **RequiredSize* contains the required buffer length.
STATUS_OBJECT_NAME_NOT_FOUND	The specified device property was not found.

Remarks

Kernel-mode drivers use the **IoGetDevicePropertyData** routine to retrieve device properties that are defined as part of the unified device property model. For more information about device properties, see Device Properties.

Drivers can use the IoSetDevicePropertyData routine to modify a device property.

Callers of **IoGetDevicePropertyData** must be running at IRQL <= APC_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DEVPROPKEY

DEVPROPTYPE

IoSetDevicePropertyData

IoGetDmaAdapter function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDmaAdapter** routine returns a pointer to the DMA adapter structure for a physical device object.

Syntax

```
_DMA_ADAPTER * IoGetDmaAdapter(
  PDEVICE_OBJECT      PhysicalDeviceObject,
  _DEVICE_DESCRIPTION *DeviceDescription,
  PULONG              NumberOfMapRegisters
);
```

Parameters

`PhysicalDeviceObject`

Pointer to the physical device object for the device requesting the DMA adapter structure.

`DeviceDescription`

Pointer to a DEVICE_DESCRIPTION structure, which describes the attributes of the physical device.

`NumberOfMapRegisters`

A pointer to, on output, the maximum number of map registers that the driver can allocate for any DMA transfer operation.

Return Value

IoGetDmaAdapter returns a pointer to a DMA_ADAPTER structure, which contains pointers to functions that support system-defined DMA operations. If the structure cannot be allocated, the routine returns **NULL**.

Remarks

Before calling this routine, a driver must zero-initialize the DEVICE_DESCRIPTION structure pointed to by *DeviceDescription* and then add the relevant information for its device to this structure.

On success, the **DmaOperations** member of the routine's return value points to a **DMA_ADAPTER** structure. This structure contains a pointer to a DMA_OPERATIONS structure, which is a table of pointers to functions that the driver can subsequently use to perform DMA operations. The version of this structure that the routine returns is determined as follows:

- If the driver sets the **Version** member of the **DEVICE_DESCRIPTION** structure to DEVICE_DESCRIPTION_VERSION or DEVICE_DESCRIPTION_VERSION1, the returned **DMA_ADAPTER** structure points to version 1 of the **DMA_OPERATIONS** structure.
- If the driver sets **Version** = DEVICE_DESCRIPTION_VERSION2, the returned **DMA_ADAPTER** structure points to version 2 of the **DMA_OPERATIONS** structure if version 2 is supported; otherwise, the routine returns **NULL**. Drivers must check to see if version 2 is supported before attempting to use any version 2 function.
- If the driver sets **Version** = DEVICE_DESCRIPTION_VERSION3, the returned **DMA_ADAPTER** structure points to version 3 of the **DMA_OPERATIONS** structure if version 3 is supported; otherwise, the routine

returns **NULL**. Drivers must check to see if version 3 is supported before attempting to use any version 3 function. Version 3 is supported starting with Windows 8.

A PnP driver calls **IoGetDmaAdapter** when its AddDevice routine is called or when it handles a PnP IRP_MN_START_DEVICE request for a device. This IRP includes information about the device's hardware resources that the driver must supply in the *DeviceDescription* structure.

The caller uses the **MaximumLength** member in the *DeviceDescription* structure to indicate the optimal number of map registers it can use. The I/O manager will attempt to allocate enough map registers to accommodate a DMA transfer operation of this maximum size. On output, the I/O manager returns, in the *NumberOfMapRegisters* parameter, the number of map registers that it allocates. Drivers should check the returned value; there is no guarantee a driver will receive the same number of map registers it requested.

To free the adapter object, the driver should call PutDmaAdapter through the pointer returned in the **DMA_ADAPTER** structure.

> **Note** As previously described, **IoGetDmaAdapter** returns **NULL** if it does not support the version of the **DMA_ADAPTER** structure that is specified by *DeviceDescription*->**Version**. Callers should rely on this behavior to determine whether the routine returns a pointer to the requested version of the **DMA_ADAPTER** structure. When **IoGetDmaAdapter** returns a pointer to version 1 or version 2 of the **DMA_ADAPTER** structure, the **Version** member of this structure is always set to 1. Thus, the caller cannot use the **Version** member of the returned **DMA_ADAPTER** structure to distinguish between versions 1 and 2 of this structure.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

AddDevice

DEVICE_DESCRIPTION

DMA_ADAPTER

DMA_OPERATIONS

IRP_MN_START_DEVICE

PutDmaAdapter

IoGetDriverObjectExtension function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetDriverObjectExtension** routine retrieves a previously allocated per-driver context area.

Syntax

```
__drv_aliasesMem PVOID IoGetDriverObjectExtension(
  PDRIVER_OBJECT DriverObject,
  PVOID          ClientIdentificationAddress
);
```

Parameters

`DriverObject`

Specifies the driver object with which the context area is associated.

`ClientIdentificationAddress`

Specifies the unique identifier, provided when it was allocated, of the context area to be retrieved.

Return Value

IoGetDriverObjectExtension returns a pointer to the context area, if any or returns **NULL**.

Remarks

Drivers call **IoGetDriverObjectExtension** to retrieve a pointer to a previously allocated extension area.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoAllocateDriverObjectExtension

IoGetFunctionCodeFromCtlCode macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetFunctionCodeFromCtlCode** macro returns the value of the function code contained in an I/O control code.

Syntax

```
void IoGetFunctionCodeFromCtlCode(
   ControlCode
);
```

Parameters

`ControlCode`

The IOCTL_*XXX* (or FSCTL_*XXX*) value, which can be obtained from the driver's I/O stack location of the IRP at **Parameters.DeviceIoControl.IoControlCode**.

Return Value

IoGetFunctionCodeFromCtlCode returns the value of the Function part of the given IOCTL_XXX code.

Remarks

For information about the layout of IOCTL codes, see Using I/O Control Codes.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IoBuildDeviceIoControlRequest

IoGetInitialStack function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetInitialStack** routine returns the base address of the current thread's stack.

Syntax

```
PVOID IoGetInitialStack(

);
```

Parameters

This function has no parameters.

Return Value

IoGetInitialStack returns the initial base address of the current thread's stack.

Remarks

Highest-level drivers can call this routine, particularly file systems attempting to determine whether they've been passed a pointer to a location on the current thread's stack.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlIoApcLte, HwStorPortProhibitedDDIs

See Also

IoGetRemainingStackSize

IoGetStackLimits

IoGetIoPriorityHint function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetIoPriorityHint** routine gets the priority hint value from an IRP.

Syntax

```
IO_PRIORITY_HINT IoGetIoPriorityHint(
  PIRP Irp
);
```

Parameters

`Irp`

Specifies the IRP to obtain the priority hint from.

Return Value

IoGetIoPriorityHint returns the IO_PRIORITY_HINT value that indicates the current priority hint.

Remarks

For more information about priority hints, see Using IRP Priority Hints.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_PRIORITY_HINT

IoSetIoPriorityHint

IoGetNextIrpStackLocation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetNextIrpStackLocation** routine gives a higher level driver access to the next-lower driver's I/O stack location in an IRP so the caller can set it up for the lower driver.

Syntax

```
__drv_aliasesMem PIO_STACK_LOCATION IoGetNextIrpStackLocation(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to the IRP.

Return Value

IoGetNextIrpStackLocation returns a pointer to the next-lower-level driver's I/O stack location in the given IRP.

Remarks

Each driver that passes IRPs on to lower drivers must set up the stack location for the next lower driver. A driver calls **IoGetNextIrpStackLocation** to get a pointer to the next-lower driver's I/O stack location.

If a driver is passing the same parameters that it received to the next-lower driver, it should call IoCopyCurrentIrpStackLocationToNext or IoSkipCurrentIrpStackLocation instead of getting a pointer to the next-lower stack location and copying the parameters manually.

The return value is a pointer to an IO_STACK_LOCATION structure. For more information, see I/O Stack Locations.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IO_STACK_LOCATION

IoCallDriver

IoCopyCurrentIrpStackLocationToNext

IoGetCurrentIrpStackLocation

IoSetNextIrpStackLocation

IoSkipCurrentIrpStackLocation

IoGetRelatedDeviceObject function

1/25/2019 • 2 minutes to read • Edit Online

Given a file object, the **IoGetRelatedDeviceObject** routine returns a pointer to the corresponding device object.

Syntax

```
PDEVICE_OBJECT IoGetRelatedDeviceObject(
  PFILE_OBJECT FileObject
);
```

Parameters

`FileObject`

Pointer to the file object.

Return Value

IoGetRelatedDeviceObject returns a pointer to the device object.

Remarks

When called on a file object that represents the underlying storage device, **IoGetRelatedDeviceObject** returns the highest-level device object in the storage device stack. To obtain the highest-level device object in the file system driver stack, drivers must call **IoGetRelatedDeviceObject** on a file object that represents the file system's driver stack, and the file system must currently be mounted. (Otherwise, the storage device stack is traversed instead of the file system stack.)

To ensure that the file system is mounted on the storage device, the driver must have specified an appropriate access mask, such as FILE_READ_DATA or FILE_WRITE_ATTRIBUTES, when opening the file or device represented by the file object. Specifying FILE_READ_ATTRIBUTES does not cause the file system to be mounted.

The caller must be running at IRQL <= DISPATCH_LEVEL. Usually, callers of this routine are running at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

IoGetDeviceObjectPointer

IoGetRemainingStackSize function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetRemainingStackSize** routine returns the current amount of available kernel-mode stack space.

Syntax

```
ULONG_PTR IoGetRemainingStackSize(

);
```

Parameters

This function has no parameters.

Return Value

IoGetRemainingStackSize returns the number of bytes of stack space in the current thread context.

Remarks

Highest-level drivers, such as file systems, can call this routine, particularly drivers that use recursive code paths. Such a driver would call **IoGetRemainingStackSize** before launching a recursion to determine whether it should continue processing on an alternate code path.

For Windows Server 2003 Service Pack 1 (SP1) and later versions of Windows, callers of **IoGetRemainingStackSize** can be running at any IRQL. For earlier versions of Windows, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	See Remarks section.

See Also

IoGetInitialStack

IoGetStackLimits

IoGetStackLimits function

1/25/2019 • 2 minutes to read • Edit Online

The **IoGetStackLimits** routine returns the boundaries of the current thread's stack frame.

Syntax

```
void IoGetStackLimits(
  PULONG_PTR LowLimit,
  PULONG_PTR HighLimit
);
```

Parameters

`LowLimit`

Pointer to a caller-supplied variable in which this routine returns the lower offset of the current thread's stack frame.

`HighLimit`

Pointer to a caller-supplied variable in which this routine returns the higher offset of the current thread's stack frame.

Return Value

None

Remarks

Highest-level drivers can call this routine, particularly file systems that have been passed a pointer to a location on the current thread's stack.

In Windows Server 2003 Service Pack 1 (SP1) and later versions of Windows, callers of **IoGetStackLimits** can be running at any IRQL. For earlier operating systems, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	See Remarks section.

See Also

IoGetInitialStack

IoGetRemainingStackSize

IoInitializeDpcRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInitializeDpcRequest** routine registers a driver-supplied DpcForIsr routine.

Syntax

```
void IoInitializeDpcRequest(
  PDEVICE_OBJECT  DeviceObject,
  PIO_DPC_ROUTINE DpcRoutine
);
```

Parameters

`DeviceObject`

Supplies a pointer to the DEVICE_OBJECT structure representing the physical device that generates interrupts.

`DpcRoutine`

Supplies a pointer to the driver-supplied *DpcForIsr* routine.

Return Value

None

Remarks

IoInitializeDpcRequest associates a driver-supplied DpcForIsr routine with a given device object. The driver's InterruptService routine (ISR) can call IoRequestDpc to queue the *DpcForIsr* routine. The *DpcForIsr* routine completes interrupt-driven I/O operations at a lower IRQL than that of the ISR.

PnP drivers call **IoInitializeDpcRequest** from the AddDevice routine.

For more information, see DPC Objects and DPCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

AddDevice

DEVICE_OBJECT

DpcForIsr

InterruptService

IoRequestDpc

KeInitializeDpc

IoInitializeIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInitializeIrp** routine initializes a given IRP that was allocated by the caller.

Syntax

```
void IoInitializeIrp(
  PIRP   Irp,
  USHORT PacketSize,
  CCHAR  StackSize
);
```

Parameters

`Irp`

Pointer to the IRP to be initialized.

`PacketSize`

Specifies the size in bytes of the IRP.

`StackSize`

Specifies the number of stack locations in the IRP.

Return Value

None

Remarks

Drivers use **IoInitializeIrp** to initialize IRPs the driver allocated as raw memory. Do not use **IoInitializeIrp** to initialize an IRP allocated by **IoAllocateIrp**. **IoAllocateIrp** automatically initializes the members of the IRP.

Drivers can use **IoInitializeIrp** to reinitialize an IRP for reuse only under certain circumstances. See Reusing IRPs for details.

If the driver associates an MDL with the IRP it allocated, the driver is responsible for releasing the MDL when the IRP is completed.

An intermediate or highest-level driver also can call **IoBuildDeviceIoControlRequest**, **IoBuildAsynchronousFsdRequest**, or **IoBuildSynchronousFsdRequest** to set up requests it sends to lower-level drivers. Only a highest-level driver can call **IoMakeAssociatedIrp**.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IoReuseIrp, HwStorPortProhibitedDDIs

See Also

IRP

IoAllocateIrp

IoAllocateMdl

IoBuildPartialMdl

IoFreeIrp

IoFreeMdl

IoReuseIrp

IoSetNextIrpStackLocation

IoSizeOfIrp

IoInitializeRemoveLock macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoInitializeRemoveLock** routine initializes a remove lock for a device object.

Syntax

```
void IoInitializeRemoveLock(
   Lock,
   Tag,
   Maxmin,
   HighWater
);
```

Parameters

`Lock`

Pointer to a caller-supplied **IO_REMOVE_LOCK** structure that this routine initializes with information about the lock, including a counter and a synchronization event. A driver writer must allocate this structure as part of the device object's device extension.

`Tag`

Specifies a tag to identify the creator of the lock. Driver writers typically use a 4-character string, specified in reverse order, like the tags used for ExAllocatePoolWithTag.

The I/O system uses this parameter if Driver Verifier is enabled, and on checked builds regardless of whether Driver Verifier is enabled. The caller should always supply a nonzero tag value for this parameter, for both free and checked builds.

`Maxmin`

`HighWater`

Return Value

None

Remarks

A driver can use a remove lock to track outstanding I/O operations on a device and to determine when the driver can delete its device object in response to an IRP_MN_REMOVE_DEVICE request.

Before calling **IoInitializeRemoveLock**, a driver should allocate an **IO_REMOVE_LOCK** structure in its device extension. A driver typically calls **IoInitializeRemoveLock** in its AddDevice routine, when the driver initializes the rest of the device extension for a device object.

After the **IoReleaseRemoveLockAndWait** routine returns, the driver should consider the device to be in a state in which it is ready to be removed and cannot perform I/O operations. Therefore, the driver must not call **IoInitializeRemoveLock** to re-initialize the remove lock. Violation of this rule while the driver is being verified by Driver Verifier will result in a bug check.

Because the driver stores the **IO_REMOVE_LOCK** structure in the device extension of a device object, the remove lock is deleted when the driver deletes the device extension as part of processing an IRP_MN_REMOVE_DEVICE request.

For more information, see Using Remove Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

IoAcquireRemoveLock

IoInitializeRemoveLock

IoReleaseRemoveLock

IoReleaseRemoveLockAndWait

IoInitializeTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInitializeTimer** routine sets up a driver-supplied IoTimer routine associated with a given device object.

Syntax

```
NTSTATUS IoInitializeTimer(
  PDEVICE_OBJECT         DeviceObject,
  PIO_TIMER_ROUTINE      TimerRoutine,
  __drv_aliasesMem PVOID Context
);
```

Parameters

`DeviceObject`

Pointer to a device object representing a device on which I/O operations can time out.

`TimerRoutine`

Pointer to the driver-supplied *IoTimer* routine.

`Context`

Pointer to the driver-determined context with which its *IoTimer* routine will be called.

Return Value

IoInitializeTimer returns STATUS_SUCCESS if the *IoTimer* routine is set up.

Remarks

IoInitializeTimer should be called only once per device object.

A driver's IoTimer routine is called once per second after the driver enables the timer by calling IoStartTimer. The driver can disable the timer by calling IoStopTimer and can reenable it again with **IoStartTimer**.

The driver's *IoTimer* routine is called at IRQL = DISPATCH_LEVEL and therefore must not contain pageable code.

When the timer is running, the I/O manager calls the driver-supplied *IoTimer* routine once per second. Drivers whose time-out routines should be called at variable intervals or at intervals of finer granularity can set up a CustomTimerDpc routine and use the **Ke*Xxx*Timer** routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoStartTimer

IoStopTimer

IoTimer

KeInitializeTimer

KeSetTimer

IoInitializeWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInitializeWorkItem** routine initializes a work item that the caller has already allocated.

Syntax

```
void IoInitializeWorkItem(
  PVOID        IoObject,
  PIO_WORKITEM IoWorkItem
);
```

Parameters

`IoObject`

Pointer to the caller's driver object or to one of the caller's device objects. If the caller will later pass the work item to IoQueueWorkItem, *IoObject* must point to a device object.

`IoWorkItem`

Pointer to the IO_WORKITEM structure to initialize.

Return Value

None

Remarks

The driver allocates its own storage for an IO_WORKITEM structure, and then calls **IoInitializeWorkItem** to initialize it. Memory must be allocated from nonpaged pool. To determine the number of bytes that is necessary to hold an **IO_WORKITEM** structure, call IoSizeofWorkItem.

An **IO_WORKITEM** structure that is initialized by **IoInitializeWorkItem**must be uninitialized by IoUninitializeWorkItem before it is freed.

A related routine, IoAllocateWorkItem, both allocates and initializes a work item. By combining allocation and initialization into a single call, **IoAllocateWorkItem** might be more convenient for some drivers to use than **IoInitializeWorkItem**, which does initialization only. However, **IoInitializeWorkItem** is required by drivers that must initialize work items in previously allocated storage.

For example, a driver might allocate a block of storage for a pool of work items. The driver can call **IoInitializeWorkItem** to initialize a work item when the work item is allocated from the pool. The driver can call **IoUninitializeWorkItem** to uninitialize the work item when it is returned to the pool. During the initial storage allocation for the pool, the driver can call **IoSizeofWorkitem** to determine how much storage to allocate for each work item.

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_WORKITEM

IoAllocateWorkItem

IoQueueWorkItem

IoQueueWorkItemEx

IoSizeofWorkItem

IoUninitializeWorkItem

IoInvalidateDeviceRelations function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInvalidateDeviceRelations** routine notifies the PnP manager that the relations for a device (such as bus relations, ejection relations, removal relations, and the target device relation) have changed.

Syntax

```
void IoInvalidateDeviceRelations(
  PDEVICE_OBJECT       DeviceObject,
  DEVICE_RELATION_TYPE Type
);
```

Parameters

`DeviceObject`

Pointer to the PDO for the device.

`Type`

Specifies a **DEVICE_RELATION_TYPE** enumeration value that describes the type of relations that have changed. Possible values include **BusRelations**, **EjectionRelations**, **RemovalRelations**, and **TargetDeviceRelation**. Starting with Windows 7, the **PowerRelations** value is also supported.

Return Value

None

Remarks

For some relation types, such as **BusRelations**, this routine causes the PnP manager or power manager to gather updated relations information by sending an IRP_MN_QUERY_DEVICE_RELATIONS request to the drivers for the device. For other relation types, such as **EjectionRelations**, the PnP manager does not need to gather new relation information immediately; the PnP manager queries drivers for ejection relations only when it is preparing to eject a device.

After a bus driver calls **IoInvalidateDeviceRelations** to inform the PnP manager that a device has disappeared, the bus driver must continue to handle PnP IRPs for that device until it receives an IRP_MN_REMOVE_DEVICE request. In response to this IRP, the bus driver returns STATUS_NO_SUCH_DEVICE. Until the bus driver successfully completes the **IRP_MN_REMOVE_DEVICE** request, the bus driver can access the device extension to check its flags for the device.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IRP_MN_QUERY_DEVICE_RELATIONS

IRP_MN_REMOVE_DEVICE

IoInvalidateDeviceState function

1/25/2019 • 2 minutes to read • Edit Online

The **IoInvalidateDeviceState** routine notifies the PnP manager that some aspect of the PnP state of a device has changed.

Syntax

```
void IoInvalidateDeviceState(
  PDEVICE_OBJECT PhysicalDeviceObject
);
```

Parameters

`PhysicalDeviceObject`

Pointer to the PDO for the device.

Return Value

None

Remarks

Drivers call this routine to indicate that something has changed with respect to one of the following aspects of a device's PnP state:

PNP_DEVICE_DISABLED

PNP_DEVICE_DONT_DISPLAY_IN_UI

PNP_DEVICE_FAILED

PNP_DEVICE_NOT_DISABLEABLE

PNP_DEVICE_REMOVED

PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED

For descriptions of the preceding constants, see PNP_DEVICE_STATE.

In response to this routine, the PnP manager sends an IRP_MN_QUERY_PNP_DEVICE_STATE request to the device stack, to determine the current PnP state of the device.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IRP_MN_QUERY_PNP_DEVICE_STATE

PNP_DEVICE_STATE

IoIs32bitProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoIs32bitProcess** routine checks whether the originator of the current I/O request is a 32-bit user-mode application.

Syntax

```
BOOLEAN IoIs32bitProcess(
  PIRP Irp
);
```

Parameters

`Irp`

Optional pointer to the IRP containing the current I/O request. Note that this must be the original IRP that was issued by the I/O manager. If the caller is a fast I/O dispatch routine (and thus there is no IRP), the caller should set this parameter to **NULL**.

Return Value

IoIs32bitProcess returns **TRUE** if the originator of the current I/O request is a 32-bit user-mode process. Otherwise, it returns **FALSE**.

Remarks

Drivers call **IoIs32bitProcess** to determine whether an I/O request is likely to contain data elements that need to be converted, or "thunked," before they can be used in a 64-bit driver.

If the *Irp* parameter is **NULL**, and the caller is running in the context of a user-mode process, **IoIs32bitProcess** returns **TRUE** if this process is a 32-bit process.

If the *Irp* parameter is not **NULL**, it must point to an IRP that was issued by the I/O manager on behalf of a user-mode process. **IoIs32bitProcess** cannot be used to check driver-allocated IRPs. Driver-allocated IRPs, as well as kernel-mode drivers, are assumed to be 64-bit-ready.

For more information, see Programming Issues for 64-Bit Drivers.

Requirements

Minimum supported client	Available in 64-bit versions of Microsoft Windows XP and later 64-bit versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

IoIsErrorUserInduced macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoIsErrorUserInduced** routine determines whether an I/O error encountered while processing a request to a removable-media device was caused by the user.

Syntax

```
void IoIsErrorUserInduced(
   Status
);
```

Parameters

`Status`

Specifies the current NTSTATUS value, usually within the driver's DpcForIsr routine.

Return Value

IoIsErrorUserInduced returns TRUE if an I/O request failed because of a user-induced error.

Remarks

This routine indicates whether an I/O request failed for one of the following user-correctable conditions:

STATUS_DEVICE_NOT_READY

STATUS_IO_TIMEOUT

STATUS_MEDIA_WRITE_PROTECTED

STATUS_NO_MEDIA_IN_DEVICE

STATUS_UNRECOGNIZED_MEDIA

STATUS_VERIFY_REQUIRED

STATUS_WRONG_VOLUME

If **IoIsErrorUserInduced** returns **TRUE**, the removable-media driver must call IoSetHardErrorOrVerifyDevice before completing the IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

IRQL	Any level

See Also

IoAllocateErrorLogEntry

IoSetHardErrorOrVerifyDevice

IoWriteErrorLogEntry

IoIsWdmVersionAvailable function

1/25/2019 • 2 minutes to read • Edit Online

The **IoIsWdmVersionAvailable** routine checks whether a given WDM version is supported by the operating system.

Syntax

```
BOOLEAN IoIsWdmVersionAvailable(
  UCHAR MajorVersion,
  UCHAR MinorVersion
);
```

Parameters

`MajorVersion`

Specifies the major version number of WDM that is requested.

`MinorVersion`

Specifies the minor version number of WDM that is requested.

Return Value

IoIsWdmVersionAvailable returns **TRUE** if the version of WDM that the operating system provides is greater than or equal to the version number of WDM being requested. Otherwise, it returns **FALSE**.

Remarks

Drivers should use the RtlIsNtDdiVersionAvailable routine instead of the **IoIsWdmVersionAvailable** routine.

Cross-platform drivers should use this routine to check the WDM version before performing any operations that vary by platform or are not supported in all versions of WDM.

The WDM_MAJORVERSION and WDM_MINORVERSION constants, which are defined in the Wdm.h header file, specify the WDM major and minor version numbers for the current version of Windows. The following lists the WDM version provided with each operating system.

OPERATING SYSTEM	WDM MAJOR VERSION	WDM MINOR VERSION
Windows 7	6	0x00
Windows Server 2008 R2	6	0x00
Windows Server 2008	6	0x00
Windows Vista	6	0x00
Windows Server 2003	1	0x30

Windows XP	1	0x20
Windows 2000	1	0x10
Windows Me	1	0x05
Windows 98	1	0x00

Note that the minor version number is defined as a hexadecimal value.

Later versions of WDM support all the features available in earlier versions of WDM; that is, each version of WDM is a superset of the previous WDM version.

The following call returns **TRUE** on any of the listed operating systems, because all these systems support all the features of WDM 1.0:

```
bVersion = IoIsWdmVersionAvailable(1,0);
```

The following example shows how a driver can dynamically detect the current operating system:

```
if (IoIsWdmVersionAvailable(1, 0x10)) {
    //
    //If WDM 1.10 is supported, this is Windows 2000
    //or better.
    //
} else if (IoIsWdmVersionAvailable(1, 5)) {
    //
    //If WDM 1.05 is supported, this is Windows ME
    //or better.
    //
} else {
    //
    //WDM 1.0 is always supported, so this is Windows 98,
    //Windows 98 SE, or better.
    //
}
```

As the example shows, calling **IoIsWdmVersionAvailable**(1, 5) returns **TRUE** on Windows Me, Windows 2000, and any succeeding operating systems, but **FALSE** on Windows 98 and Windows 98 SE.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

RtlIsNtDdiVersionAvailable

IoMarkIrpPending function

1/25/2019 • 2 minutes to read • Edit Online

The **IoMarkIrpPending** routine marks the specified IRP, indicating that a driver's dispatch routine subsequently returned STATUS_PENDING because further processing is required by other driver routines.

Syntax

```
void IoMarkIrpPending(
  PIRP Irp
);
```

Parameters

`Irp`

Pointer to the IRP to be marked as pending.

Return Value

None

Remarks

Unless the driver's dispatch routine completes the IRP (by calling IoCompleteRequest) or passes the IRP on to lower drivers, it must call **IoMarkIrpPending** with the IRP. Otherwise, the I/O manager attempts to complete the IRP as soon as the dispatch routine returns control.

After calling **IoMarkIrpPending**, the dispatch routine must return STATUS_PENDING, even if some routine completes the IRP (by calling IoCompleteRequest) before the dispatch routine that called **IoMarkIrpPending** returns.

If a driver queues incoming IRPs, it should call **IoMarkIrpPending** before it queues each IRP. Otherwise, an IRP could be dequeued, completed by another driver routine, and freed by the system before the call to **IoMarkIrpPending** occurs, thereby causing a crash.

If a driver sets an IoCompletion routine for an IRP and then passes the IRP down to a lower driver, the *IoCompletion* routine should check the **IRP->PendingReturned** flag. If the flag is set, the *IoCompletion* routine must call **IoMarkIrpPending** with the IRP. (*IoCompletion* routines do not return STATUS_PENDING, however. For more information, see Implementing an IoCompletion Routine.)

> **Note** Drivers that create the IRP and send it to another stack must not call **IoMarkIrpPending** in their completion routine. This call will corrupt the pool header of the next allocation because there is no stack location for those drivers.

A driver that passes down the IRP and then waits on an event should not mark the IRP pending. Instead, its *IoCompletion* routine should signal the event and return STATUS_MORE_PROCESSING_REQUIRED.

If your driver calls IoSkipCurrentIrpStackLocation, be careful not to modify the IO_STACK_LOCATION structure in a way that could unintentionally affect the lower driver or the system's behavior with respect to that driver. In particular, your driver should not modify the **IO_STACK_LOCATION** structure's **Parameters** union, and should

not call **IoMarkIrpPending**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level
DDI compliance rules	CompleteRequestStatusCheck, CompletionEventChecking, IrpCancelField, LowerDriverReturn, MarkDevicePower, MarkingInterlockedQueuedIrps, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequest3

See Also

IO_STACK_LOCATION

IRP

IoCompleteRequest

IoSkipCurrentIrpStackLocation

IoStartPacket

IoOpenDeviceInterfaceRegistryKey function

1/25/2019 • 2 minutes to read • Edit Online

The **IoOpenDeviceInterfaceRegistryKey** routine returns a handle to a registry key for storing information about a particular device interface instance.

Syntax

```
NTSTATUS IoOpenDeviceInterfaceRegistryKey(
  PUNICODE_STRING SymbolicLinkName,
  ACCESS_MASK     DesiredAccess,
  PHANDLE         DeviceInterfaceRegKey
);
```

Parameters

`SymbolicLinkName`

Pointer to a string identifying the device interface instance. This string was obtained from a previous call to IoGetDeviceInterfaces, IoGetDeviceInterfaceAlias, or IoRegisterDeviceInterface.

`DesiredAccess`

Specifies the ACCESS_MASK value that represents the access the caller requires to the key, such as KEY_READ, KEY_WRITE, or KEY_ALL_ACCESS. See ZwCreateKey for a description of each KEY_*XXX* access right.

`DeviceInterfaceRegKey`

Pointer to a returned handle to the requested registry key if the call is successful.

Return Value

IoOpenDeviceInterfaceRegistryKey returns STATUS_SUCCESS if the call was successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_OBJECT_NAME_NOT_FOUND	The routine was not able to locate a registry key for the device interface instance, probably due to an error in the *SymbolicLinkName*.

STA TUS _OB JEC T_P AT H_N OT_ FOU ND	The routine was not able to locate a registry key for the device interface instance, probably due to an error in the *SymbolicLinkName*.
STA TUS _IN VAL ID_ PAR AM ETE R	Possibly indicates an error in the *SymbolicLinkName*.

Remarks

IoOpenDeviceInterfaceRegistryKey opens a nonvolatile subkey of the registry key for the device interface instance specified by *SymbolicLinkName*. Drivers can store information in this subkey that is specific to this instance of the device interface, such as the default resolution for a camera. User-mode applications can access this subkey using **SetupDi*Xxx*** routines.

The driver must call ZwClose to close the handle returned from this routine when access is no longer required.

Callers of **IoOpenDeviceInterfaceRegistryKey** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

IoGetDeviceInterfaceAlias

IoGetDeviceInterfaces

IoRegisterDeviceInterface

ZwClose

IoOpenDeviceRegistryKey function

1/25/2019 • 2 minutes to read • Edit Online

The **IoOpenDeviceRegistryKey** routine returns a handle to a device-specific or a driver-specific registry key for a particular device instance.

Syntax

```
NTSTATUS IoOpenDeviceRegistryKey(
  PDEVICE_OBJECT DeviceObject,
  ULONG          DevInstKeyType,
  ACCESS_MASK    DesiredAccess,
  PHANDLE        DeviceRegKey
);
```

Parameters

`DeviceObject`

Pointer to the PDO of the device instance for which the registry key is to be opened.

`DevInstKeyType`

Specifies flags indicating whether to open a device-specific hardware key or a driver-specific software key. The flags also indicate whether the key is relative to the current hardware profile. For more information about hardware and software keys, see Registry Keys for Drivers and Introduction to Registry Keys for Drivers.

The flags are defined as follows:

PLUGPLAY_REGKEY_DEVICE

Open the **Device Parameters** subkey under the device's hardware key (see HKLM\SYSTEM\CurrentControlSet\Enum Registry Tree), a device-specific registry subkey that contains information about the device. The key is located under the key for the device instance specified by *DeviceObject*. This flag cannot be specified with PLUGPLAY_REGKEY_DRIVER.

PLUGPLAY_REGKEY_DRIVER

Open a *software key* for storing driver-specific information. This flag cannot be specified with PLUGPLAY_REGKEY_DEVICE.

A driver's software key is also called its *driver key* because the registry contains a software key for each driver. The registry contains a list of all of the device classes, and each driver's software key resides under its device class entry. The system stores information about each driver under its software key. For more information about software keys, see The HKLM\SYSTEM\CurrentControlSet\Control Tree.

PLUGPLAY_REGKEY_CURRENT_HWPROFILE

Open a key relative to the current hardware profile for device or driver information. This allows the driver to access configuration information that is hardware-profile-specific. The caller must specify either PLUGPLAY_REGKEY_DEVICE or PLUGPLAY_REGKEY_DRIVER with this flag. For more information, see HKLM\SYSTEM\CurrentControlSet\HardwareProfiles Registry Tree.

`DesiredAccess`

Specifies the ACCESS_MASK value that represents the access the caller needs to the key. See the ZwCreateKey

routine for a description of each KEY_*XXX* access right.

`DeviceRegKey`

Pointer to a caller-allocated buffer that, on successful return, contains a handle to the requested registry key.

Return Value

IoOpenDeviceRegistryKey returns STATUS_SUCCESS if the call was successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	Possibly indicates that the caller specified an illegal set of *DevInstKeyType* flags.
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that the *DeviceObject* is not a valid PDO.

Remarks

The driver must call ZwClose to close the handle returned from this routine when access is no longer required.

The registry keys opened by this routine are nonvolatile.

User-mode setup applications can access these registry keys by using device installation functions such as SetupDiOpenDevRegKey. An example of a user-mode setup applications would be a class installer, a Win32 DLL that performs installation operations for devices in a particular Device Setup Class.

To create registry keys, use INF AddReg directives in an INF file or use SetupDiCreateDevRegKey in a setup application.

Callers of **IoOpenDeviceRegistryKey** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Introduction to Registry Keys for Drivers

ACCESS_MASK

ZwCreateKey

ZwClose

IoQueueWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoQueueWorkItem** routine associates a WorkItem routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.

Syntax

```
void IoQueueWorkItem(
  __drv_aliasesMem PIO_WORKITEM IoWorkItem,
  PIO_WORKITEM_ROUTINE          WorkerRoutine,
  WORK_QUEUE_TYPE               QueueType,
  __drv_aliasesMem PVOID        Context
);
```

Parameters

`IoWorkItem`

Pointer to an IO_WORKITEM structure that was allocated by IoAllocateWorkItem or initialized by IoInitializeWorkItem.

`WorkerRoutine`

Pointer to a WorkItem routine.

`QueueType`

Specifies a WORK_QUEUE_TYPE value that stipulates the type of system worker thread to handle the work item. Drivers must specify **DelayedWorkQueue**.

`Context`

Specifies driver-specific information for the work item. The system passes this value as the *Context* parameter to WorkItem.

Return Value

None.

Remarks

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_WORKITEM

IoAllocateWorkItem

IoQueueWorkItemEx

WorkItem

IoQueueWorkItemEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoQueueWorkItemEx** routine associates a WorkItemEx routine with a work item, and it inserts the work item into a queue for later processing by a system worker thread.

Syntax

```
void IoQueueWorkItemEx(
  __drv_aliasesMem PIO_WORKITEM IoWorkItem,
  PIO_WORKITEM_ROUTINE_EX       WorkerRoutine,
  WORK_QUEUE_TYPE               QueueType,
  __drv_aliasesMem PVOID        Context
);
```

Parameters

`IoWorkItem`

Pointer to an IO_WORKITEM structure that was allocated by IoAllocateWorkItem or initialized by IoInitializeWorkItem.

`WorkerRoutine`

Pointer to a WorkItemEx routine.

`QueueType`

Specifies a WORK_QUEUE_TYPE value that stipulates the type of system worker thread to handle the work item. Drivers must specify **DelayedWorkQueue**.

`Context`

Specifies driver-specific information for the work item. The system passes this value as the *Context* parameter to WorkItemEx.

Return Value

None

Remarks

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_WORKITEM

IoAllocateWorkItem

IoInitializeWorkItem

IoQueueWorkItem

WorkItemEx

IoRegisterContainerNotification function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterContainerNotification** routine registers a kernel-mode driver to receive notifications about a specified class of events.

Syntax

```
NTSTATUS IoRegisterContainerNotification(
  IO_CONTAINER_NOTIFICATION_CLASS     NotificationClass,
  PIO_CONTAINER_NOTIFICATION_FUNCTION CallbackFunction,
  PVOID                               NotificationInformation,
  ULONG                               NotificationInformationLength,
  PVOID                               CallbackRegistration
);
```

Parameters

`NotificationClass`

Specifies the class of events for which the caller (driver) requests notifications. Set this parameter to the following IO_CONTAINER_NOTIFICATION_CLASS enumeration value:

- **IoSessionStateNotification**

For more information, see the following Remarks section.

`CallbackFunction`

A pointer to a callback function that is implemented by the caller (driver). The I/O manager calls this function to notify the caller when an event of the class indicated by *NotificationClass* occurs. For *NotificationClass* = **IoSessionStateNotification**, this parameter is a pointer to a caller-supplied IO_SESSION_NOTIFICATION_FUNCTION function. However, the caller should cast this function pointer to type PIO_CONTAINER_NOTIFICATION_FUNCTION to match the parameter type. For more information, see the following Remarks section.

`NotificationInformation`

A pointer to a caller-allocated buffer that contains the notification information structure for an event of the class specified by *NotificationClass*. For *NotificationClass* = **IoSessionStateNotification**, *NotificationInformation* points to an IO_SESSION_STATE_NOTIFICATION structure. The caller must fill out this structure before it calls **IoRegisterContainerNotification**. During this call, **IoRegisterContainerNotification** copies the data from this structure, and the I/O manager does not access the driver's copy of the structure after the call returns.

`NotificationInformationLength`

The size, in bytes, of the notification information structure contained in the buffer that is pointed to by *NotificationInformation*. For *NotificationClass* = **IoSessionStateNotification**, set this parameter to **sizeof**(**IO_SESSION_STATE_NOTIFICATION**).

`CallbackRegistration`

A pointer to a location into which this routine writes the address of a container notification registration object. This

object is an opaque, system object in which the I/O manager stores information about the caller's container notification registration. When notifications are no longer required, the caller cancels the registration by passing this object pointer to the IoUnregisterContainerNotification routine.

Return Value

IoRegisterContainerNotification returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER_1	Parameter *NotificationClass* is not a valid **IO_CONTAINER_NOTIFICATION_CLASS** enumeration constant.
STATUS_INVALID_PARAMETER_3	The information in the structure that is pointed to by *NotificationInformation* is incorrect.
STATUS_INVALID_PARAMETER_4	Parameter *NotificationInformationLength* does not equal the size of the notification information structure that is required for use with the specified *NotificationClass* parameter value.
STATUS_ALREADY_COMMITTED	The driver is already registered to receive *NotificationClass* notifications of events that are associated with the specified I/O object.

STA TUS _IN SUF FICI ENT _RE SO URC ES	The operating system has insufficient resources to create the requested registration.

Remarks

This routine can potentially support notifications of events in a variety of event classes. In Windows 7, this routine supports only **IoSessionStateNotification** notifications, which notify a kernel-mode driver of changes in the status of user sessions that the driver is interested in. For user-mode applications, the WTSRegisterSessionNotification function fills a similar role.

The function pointer type for the *CallbackFunction* parameter is defined as follows:

```
typedef NTSTATUS
  (*PIO_CONTAINER_NOTIFICATION_FUNCTION)(
    );
```

The caller should cast the callback function pointer to this type to match the *CallbackFunction* parameter type. **IoRegisterContainerNotification** determines the actual type of the callback function pointer from the *NotificationClass* parameter. For *NotificationClass* = **IoSessionStateNotification**, *CallbackFunction* points to an IO_SESSION_NOTIFICATION_FUNCTION function.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IO_CONTAINER_NOTIFICATION_CLASS

IO_SESSION_NOTIFICATION_FUNCTION

IO_SESSION_STATE_NOTIFICATION

IoUnregisterContainerNotification

WTSRegisterSessionNotification

IoRegisterDeviceInterface function

1/25/2019 • 3 minutes to read • Edit Online

The **IoRegisterDeviceInterface** routine registers a device interface class, if it has not been previously registered, and creates a new instance of the interface class, which a driver can subsequently enable for use by applications or other system components.

Syntax

```
NTSTATUS IoRegisterDeviceInterface(
  PDEVICE_OBJECT  PhysicalDeviceObject,
  const GUID      *InterfaceClassGuid,
  PUNICODE_STRING ReferenceString,
  PUNICODE_STRING SymbolicLinkName
);
```

Parameters

`PhysicalDeviceObject`

A pointer to the PDO for the device.

`InterfaceClassGuid`

A pointer to the class GUID that identifies the functionality (the device interface class) being registered.

`ReferenceString`

Optionally points to a UNICODE_STRING. The string must not contain any path separator characters ("/" or "\"). Function drivers typically specify **NULL** for this parameter. Filter drivers must specify **NULL**.

Reference strings are only used by a few bus drivers, such as *swenum*, which is a bus driver that uses device interface instances as placeholders for software devices created on demand. When an instance of an interface is opened, the I/O manager passes the instance's reference string to the driver. The string becomes part of the interface instance's name (as an appended path component). The driver can then use the reference string to differentiate between two interface instances of the same class for a single device.

On Microsoft Windows 98/Me systems, the *ReferenceString* value can be no longer than MAX_PATH characters. There is no length limit on Windows 2000 and later versions of Windows.

`SymbolicLinkName`

A pointer to a Unicode string structure allocated by the caller. If this routine is successful, it initializes the Unicode string and allocates the string buffer containing the kernel-mode path to the symbolic link for an instance of the specified device interface class.

The caller must treat *SymbolicLinkName* as opaque and must not disassemble it.

The caller is responsible for freeing *SymbolicLinkName* with RtlFreeUnicodeString when it is no longer needed.

Return Value

IoRegisterDeviceInterface returns STATUS_SUCCESS if the call was successful. Possible error return values include the following.

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ DEV ICE_ REQ UES T	The parameters are invalid. Possibilities include that *PhysicalDeviceObject* does not point to a valid PDO, or that the *ReferenceString* string contains an invalid character.

Remarks

IoRegisterDeviceInterface registers a device interface class, if it has not been previously registered, and creates a new instance of the interface class. A driver can call this routine several times for a given device to register several interface classes and create instances of the classes. A function or filter driver typically registers device interfaces in its AddDevice routine. For example, a fault-tolerant volume driver might register an instance of a fault-tolerant-volume interface and an instance of a volume interface for a particular volume.

If the device interface class has not been registered previously, the I/O manager creates a registry key for it, along with instance-specific persistent storage under the key. Drivers can access this persistent storage using IoOpenDeviceInterfaceRegistryKey.

A driver registers an interface instance once and then calls IoSetDeviceInterfaceState to enable and disable the interface.

Registered interfaces persist across operating system reboots. If the specified interface is already registered, the I/O manager passes its name in *SymbolicLinkName*and returns the informational success status STATUS_OBJECT_NAME_EXISTS.

Most drivers use a **NULL** reference string for a device interface instance. If a driver uses a non-**NULL** reference string, it must do additional work, including possibly managing its own namespace and security. A filter driver that exposes a device interface must use a **NULL** *ReferenceString* to avoid conflicts with other drivers in the device stack.

Callers of this routine are not required to remove the registration for a device interface when it is no longer needed. Device interface registrations can be removed from user mode, if necessary.

Callers of **IoRegisterDeviceInterface** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	IrqlIoPassive3, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoGetDeviceInterfaces

IoOpenDeviceInterfaceRegistryKey

IoSetDeviceInterfaceState

RtlFreeUnicodeString

IoRegisterLastChanceShutdownNotification function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterLastChanceShutdownNotification** routine registers a driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down, after all file systems have been flushed.

Syntax

```
NTSTATUS IoRegisterLastChanceShutdownNotification(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object of the device for which the driver requests shutdown notification. The system passes this pointer to the driver's DispatchShutdown routine.

Return Value

IoRegisterLastChanceShutdownNotification returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

The **IoRegisterLastChanceShutdownNotification** routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP for the specified device when the system shuts down. The driver receives one such IRP for each device it registers to receive notification for. Drivers handle **IRP_MJ_SHUTDOWN** IRPs within their DispatchShutdown routines.

For any device that is registered with this routine, the system sends the **IRP_MJ_SHUTDOWN** IRP after all file systems are flushed. Only one driver in a device stack should register to receive shutdown notification, by calling either IoRegisterShutdownNotification or **IoRegisterLastChanceShutdownNotification**.

If the driver ceases to require shutdown notification for that device, use IoUnregisterShutdownNotification to remove the driver from the shutdown notification queue.

A driver that calls **IoRegisterLastChanceShutdownNotification** must satisfy the following restrictions in its *DispatchShutdown* routine:

- The *DispatchShutdown* routine must not call any pageable routines.
- The *DispatchShutdown* routine must not access pageable memory.
- The *DispatchShutdown* routine must not perform any file I/O operations.

Most drivers that require shutdown notification should call the IoRegisterShutdownNotification routine, which does not impose these limitations on the *DispatchShutdown* routine, and which causes the *DispatchShutdown* routine to be called before the file systems are flushed. Only drivers that must do some cleanup after the file systems are flushed, such as a driver for a mass storage device, should use **IoRegisterLastChanceShutdownNotification**.

The registered *DispatchShutdown* routine is called before the power manager sends an IRP_MN_SET_POWER request for **PowerSystemShutdown**. The *DispatchShutdown* routine is not called for transitions to any other power states.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Not available in Microsoft Windows 98/Me.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DispatchShutdown

IoRegisterShutdownNotification

IoUnregisterShutdownNotification

IoRegisterPlugPlayNotification function

1/25/2019 • 5 minutes to read • Edit Online

The **IoRegisterPlugPlayNotification** routine registers a Plug and Play (PnP) notification callback routine to be called when a PnP event of the specified category occurs.

Syntax

```
NTSTATUS IoRegisterPlugPlayNotification(
  IO_NOTIFICATION_EVENT_CATEGORY        EventCategory,
  ULONG                                 EventCategoryFlags,
  PVOID                                 EventCategoryData,
  PDRIVER_OBJECT                        DriverObject,
  PDRIVER_NOTIFICATION_CALLBACK_ROUTINE CallbackRoutine,
  __drv_aliasesMem PVOID                Context,
  PVOID                                 *NotificationEntry
);
```

Parameters

`EventCategory`

The category of PnP event for which the callback routine is being registered. *EventCategory* must be one of the following:

EventCategoryDeviceInterfaceChange

PnP events in this category include the arrival (enabling) of a new instance of a device interface class (GUID_DEVICE_INTERFACE_ARRIVAL), or the removal (disabling) of an existing device interface instance (GUID_DEVICE_INTERFACE_REMOVAL).

EventCategoryHardwareProfileChange

PnP events in this category include query-change (GUID_HWPROFILE_QUERY_CHANGE), change-complete (GUID_HWPROFILE_CHANGE_COMPLETE), and change-cancel (GUID_HWPROFILE_CHANGE_CANCELLED) of a hardware profile.

EventCategoryTargetDeviceChange

PnP events in this category include events related to removing a device: the device's drivers received a query-remove IRP (GUID_TARGET_DEVICE_QUERY_REMOVE), the drivers completed a remove IRP (GUID_TARGET_DEVICE_REMOVE_COMPLETE), or the drivers received a cancel-remove IRP (GUID_TARGET_DEVICE_REMOVE_CANCELLED). This category is also used for custom notification events.

`EventCategoryFlags`

Flag bits that modify the registration operation. Possible values include:

PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES

Only valid with an *EventCategory* of **EventCategoryDeviceInterfaceChange**. If set, the PnP manager calls the driver callback routine for each device interface instance that is currently registered and active and registers the callback routine for future arrivals or removals of device interface instances.

`EventCategoryData`

A pointer to further information about the events for which *CallbackRoutine* is being registered. The information varies for different *EventCategory* values:

- When *EventCategory* is **EventCategoryDeviceInterfaceChange**, *EventCategoryData* must point to a GUID specifying a device interface class. *CallbackRoutine* will be called when an interface of that class is enabled or removed.
- When *EventCategory* is **EventCategoryHardwareProfileChange**, *EventCategoryData* must be **NULL**.
- When *EventCategory* is **EventCategoryTargetDeviceChange**, *EventCategoryData* must point to the file object for which PnP notification is requested.

`DriverObject`

A pointer to the caller's driver object.

To ensure that the driver remains loaded while it is registered for PnP notification, this call increments the reference count on *DriverObject*. The PnP manager decrements the reference count when this registration is removed.

`CallbackRoutine`

A pointer to the PnP notification callback routine to be called when the specified PnP event occurs.

The function prototype for this callback routine is defined as follows:

```
typedef NTSTATUS
  DRIVER_NOTIFICATION_CALLBACK_ROUTINE(
    _In_ PVOID NotificationStructure,
    _Inout_opt_ PVOID Context
    );</pre>
```

The callback routine's *NotificationStructure* is specific to the *EventCategory* value, as shown in the following table.

EVENT CATEGORY	NOTIFICATION STRUCTURE
EventCategoryDeviceInterfaceChange	DEVICE_INTERFACE_CHANGE_NOTIFICATION
EventCategoryHardwareProfileChange	HWPROFILE_CHANGE_NOTIFICATION
EventCategoryTargetDeviceChange	TARGET_DEVICE_REMOVAL_NOTIFICATION

The callback routine's *Context* parameter contains the context data the driver supplied during registration.

For information about including a function declaration for the callback routine that meets the requirements of Static Driver Verifier (SDV), see Examples.

The PnP manager calls driver callback routines at IRQL = PASSIVE_LEVEL.

`Context`

A pointer to a caller-allocated buffer containing context that the PnP manager passes to the callback routine.

`NotificationEntry`

A pointer to an opaque value returned by this call that identifies the registration. Pass this value to the IoUnregisterPlugPlayNotificationEx routine to remove the registration.

Return Value

IoRegisterPlugPlayNotification returns STATUS_SUCCESS or an appropriate error status.

Remarks

A driver registers for an event category. Each category includes one or more types of PnP events.

A driver can register different callback routines for different event categories or can register a single callback routine. A single callback routine can cast the *NotificationStructure* to a PLUGPLAY_NOTIFICATION_HEADER and use the **Event** field to determine the exact type of the notification structure.

> **NOTE**
> If the caller specifies PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES, the operating system might call the PnP notification callback routine twice for a single **EventCategoryDeviceInterfaceChange** event for an existing interface. You can safely ignore the second call to the callback. The operating system will not call the callback more than twice for a single event.

PnP notification callback routines should complete their tasks as quickly as possible and return control to the PnP manager, to prevent delays in notifying other drivers and applications that have registered for the event.

The PnP manager does not take out a reference on the file object when a driver registers for notification of an **EventCategoryTargetDeviceChange** event. If the driver's PnP notification callback routine requires access to the file object, the driver should take out an extra reference on the file object before calling **IoRegisterPlugPlayNotification**.

Typically, Kernel-Mode Driver Framework (KMDF) drivers should call **IoRegisterPlugPlayNotification** from their EvtDeviceSelfManagedIoInit callback function, and should call **IoUnregisterPlugPlayNotification** from their EvtDeviceSelfManagedIoCleanup callback function. These drivers should **not** call **IoRegisterPlugPlayNotification** from their EvtDriverDeviceAdd callback function; otherwise, the PnP notification callback routine might be called before the driver stack is started by PnP, in which case the driver will not be prepared to handle the notification.

For more information, see Using PnP Notification.

Examples

To define a PnP notification callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a PnP notification callback routine that is named `MyCallbackRoutine`, use the DRIVER_NOTIFICATION_CALLBACK_ROUTINE type as shown in this code example:

```
DRIVER_NOTIFICATION_CALLBACK_ROUTINE MyCallbackRoutine;</pre>
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyCallbackRoutine(
    PVOID NotificationStructure,
    PVOID Context
    )
  {
      // Function body
  }
```

The **DRIVER_NOTIFICATION_CALLBACK_ROUTINE** function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are

applied to the DRIVER_NOTIFICATION_CALLBACK_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about `_Use_decl_annotations_`, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION

EvtDeviceSelfManagedIoCleanup

EvtDeviceSelfManagedIoInit

EvtDriverDeviceAdd

HWPROFILE_CHANGE_NOTIFICATION

IoUnregisterPlugPlayNotification

IoUnregisterPlugPlayNotificationEx

PLUGPLAY_NOTIFICATION_HEADER

TARGET_DEVICE_CUSTOM_NOTIFICATION

TARGET_DEVICE_REMOVAL_NOTIFICATION

IoRegisterShutdownNotification function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRegisterShutdownNotification** routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP when the system is shut down.

Syntax

```
NTSTATUS IoRegisterShutdownNotification(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object of the device for which the driver requests shutdown notification. The system passes this pointer to the driver's DispatchShutdown routine.

Return Value

IoRegisterShutdownNotification returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

The **IoRegisterShutdownNotification** routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP for the specified device when the system shuts down. The driver receives one such IRP for each device it registers to receive notification for. Drivers handle **IRP_MJ_SHUTDOWN** IRPs within their DispatchShutdown routines.

If the driver ceases to require shutdown notification for the device, use IoUnregisterShutdownNotification to remove the driver from the shutdown notification queue.

Only one driver in a device stack should register to receive shutdown notification.

The system sends the driver the **IRP_MJ_SHUTDOWN** request before it flushes the file systems. Some drivers, such as drivers for mass storage devices, can require shutdown notification after the system flushes the file systems. To receive shutdown notification for a device after the file systems are flushed, use the IoRegisterLastChanceShutdownNotification routine instead.

The registered *DispatchShutdown* routine is called before the power manager sends an IRP_MN_SET_POWER request for **PowerSystemShutdown**. The *DispatchShutdown* routine is not called for transitions to any other power states.

A driver writer can make no assumptions about the order in which the driver's *DispatchShutdown* routine will be called in relation to other such routines or to other shutdown activities.

A PnP driver might register a shutdown routine to perform certain tasks before system shutdown starts, such as locking down code.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Not available in Microsoft Windows 98/Me.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DispatchShutdown

IoRegisterLastChanceShutdownNotification

IoUnregisterShutdownNotification

IoReleaseRemoveLock macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoReleaseRemoveLock** routine releases a remove lock acquired with a previous call to IoAcquireRemoveLock.

Syntax

```
void IoReleaseRemoveLock(
   RemoveLock,
   Tag
);
```

Parameters

`RemoveLock`

Pointer to an **IO_REMOVE_LOCK** structure that the caller passed to a previous call to **IoAcquireRemoveLock**.

`Tag`

Pointer to a caller-supplied tag that was passed to a previous call to **IoAcquireRemoveLock**.

If a driver specified a *Tag* when it acquired the lock, the driver must specify the same *Tag* when releasing the lock. If the tags do not match, this routine asserts on a checked build.

If the call to **IoAcquireRemoveLock** did not specify a *Tag*, then this parameter is **NULL**.

The I/O system only uses this parameter on checked builds.

Return Value

None

Remarks

A driver calls **IoReleaseRemoveLock** when it has completed the I/O operation for which it called IoAcquireRemoveLock.

- For I/O operations (including power and PnP IRPs) that set an IoCompletion routine, a driver should call **IoReleaseRemoveLock** in the *IoCompletion* routine, after calling IoCompleteRequest.
- For I/O operations that do not set an *IoCompletion* routine, a driver should call **IoReleaseRemoveLock** after passing the current IRP to the next-lower driver, but before exiting the dispatch routine.

Each call to IoAcquireRemoveLock must have a corresponding call to **IoReleaseRemoveLock**.

IoReleaseRemoveLock decrements the count of outstanding acquisitions of the remove lock. If the count goes to zero and the driver has received an IRP_MN_REMOVE_DEVICE request, **IoReleaseRemoveLock** sets an internal event. When a driver is ready to delete a device object, it calls a similar routine, IoReleaseRemoveLockAndWait. The driver makes this call only in its dispatch code for an **IRP_MN_REMOVE_DEVICE** request. The **IoReleaseRemoveLockAndWait** routine does not return until **IoReleaseRemoveLock** sets the event that indicates the acquisition count is zero. After **IoReleaseRemoveLockAndWait** returns, the driver can safely detach

and delete the device object.

For more information, see Using Remove Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	RemoveLock, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove, RemoveLockMnRemove2, RemoveLockMnSurpriseRemove, RemoveLockQueryMnRemove, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePnp, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

IoAcquireRemoveLock

IoInitializeRemoveLock

IoReleaseRemoveLockAndWait

IoReleaseRemoveLockAndWait macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoReleaseRemoveLockAndWait** routine releases a remove lock that the driver acquired in a previous call to IoAcquireRemoveLock, and waits until all acquisitions of the lock have been released.

Syntax

```
void IoReleaseRemoveLockAndWait(
   RemoveLock,
   Tag
);
```

Parameters

`RemoveLock`

Pointer to an **IO_REMOVE_LOCK** structure that the caller passed in a previous call to **IoAcquireRemoveLock**.

`Tag`

Pointer to a caller-supplied tag that was passed in a previous call to **IoAcquireRemoveLock**.

If a driver specified a *Tag* when it acquired the lock, the driver must specify the same *Tag* when releasing the lock. If the tags do not match, this routine asserts on a checked build.

If the call to **IoAcquireRemoveLock** did not specify a *Tag*, then this parameter is **NULL**.

The I/O system only uses this parameter on checked builds.

Return Value

None

Remarks

A driver typically calls this routine in its dispatch code for an IRP_MN_REMOVE_DEVICE request. To allow queued I/O requests to complete, each driver should call **IoReleaseRemoveLockAndWait** *after* it passes the remove IRP to the next-lower driver, and *before* it releases memory, calls IoDetachDevice, or calls IoDeleteDevice. The **IoReleaseRemoveLockAndWait** routine waits for the bus driver to cancel any IRPs that might be outstanding (for example, an IRP_MN_WAIT_WAKE IRP).

A driver must acquire the remove lock before calling **IoReleaseRemoveLockAndWait**. Typically, a driver calls IoAcquireRemoveLock early in its DispatchPnp routine, before the switch statement. As a result, the lock is acquired for each PnP operation, including the acquisition that is required before calling **IoReleaseRemoveLockAndWait** in the code that handles **IRP_MN_REMOVE_DEVICE**.

To release a lock from code other than the **IRP_MN_REMOVE_DEVICE** dispatch code, use IoReleaseRemoveLock.

After **IoReleaseRemoveLockAndWait** has been called for a particular remove lock, subsequent calls to **IoAcquireRemoveLock** for the same remove lock will fail. **IoReleaseRemoveLockAndWait** does not return

until all outstanding acquisitions of the remove lock have been released.

After **IoReleaseRemoveLockAndWait** returns, the driver should consider the device to be in a state in which it is ready to be removed and cannot perform I/O operations. Therefore, the driver must not call the IoInitializeRemoveLock routine to re-initialize the remove lock. Violation of this rule while the driver is being verified by Driver Verifier will result in a bug check.

For more information, see Using Remove Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IoReleaseRemoveLockAndWaitOutsideRemoveDevice, RemoveLock, RemoveLockCheck, RemoveLockMnRemove, RemoveLockMnRemove2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

DispatchPnp

IRP_MN_WAIT_WAKE

IoAcquireRemoveLock

IoDeleteDevice

IoDetachDevice

IoInitializeRemoveLock

IoReleaseRemoveLock

IoRemoveLinkShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRemoveLinkShareAccess** routine removes the access and link share-access information for a given open instance of a file object.

Syntax

```
void IoRemoveLinkShareAccess(
  PFILE_OBJECT       FileObject,
  PSHARE_ACCESS      ShareAccess,
  PLINK_SHARE_ACCESS LinkShareAccess
);
```

Parameters

`FileObject`

Pointer to the file object, which usually is being closed by the current thread.

`ShareAccess`

Pointer to the share-access structure that describes how the open file object is currently being accessed.

`LinkShareAccess`

Pointer to the share-access structure that describes how the open file object is currently being accessed.

Return Value

None

Remarks

This routine is a reciprocal to IoUpdateLinkShareAccess.

IoRemoveLinkShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoRemoveLinkShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Callers of **IoRemoveLinkShareAccess** must be running at IRQL = PASSIVE_LEVEL and in the context of the thread that requested that the *FileObject* be closed.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Target Platform	Universal
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

IoCheckLinkShareAccess

IoSetLinkShareAccess

IoUpdateLinkShareAccess

IoRemoveShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRemoveShareAccess** routine removes the access and share-access information for a given open instance of a file object.

Syntax

```
void IoRemoveShareAccess(
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess
);
```

Parameters

`FileObject`

Pointer to the file object, which usually is being closed by the current thread.

`ShareAccess`

Pointer to the share-access structure that describes how the open file object is currently being accessed.

Return Value

None

Remarks

This routine is a reciprocal to IoUpdateShareAccess.

IoRemoveShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoRemoveShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Callers of **IoRemoveShareAccess** must be running at IRQL = PASSIVE_LEVEL and in the context of the thread that requested that the *FileObject* be closed.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoCheckShareAccess

IoSetShareAccess

IoUpdateShareAccess

IoReportInterruptActive function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReportInterruptActive** routine informs the operating system that a registered interrupt service routine (ISR) is active and ready to handle interrupt requests.

Syntax

```
void IoReportInterruptActive(
  PIO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS Parameters
);
```

Parameters

`Parameters`

A pointer to an IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure that contains the connection context associated with the interrupt. The caller received this context from the IoConnectInterruptEx call that registered the ISR.

Return Value

None.

Remarks

The IoConnectInterruptEx routine registers an ISR and connects the ISR to an interrupt or interrupts. After the ISR is registered, the driver can make the ISR active or inactive by calling the **IoReportInterruptActive** or IoReportInterruptInactive routine. By default, the ISR is active after the **IoConnectInterruptEx** call.

An ISR that is in the active state can be disconnected or made inactive. To disconnect the ISR and delete its registration, call the IoDisconnectInterruptEx routine. To make the ISR inactive without changing its registration, call **IoReportInterruptInactive**.

The **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure must contain a valid connection contect obtained from an **IoConnectInterruptEx** call. Otherwise, the **IoReportInterruptActive** routine bug checks in a checked build.

For more information about **IoReportInterruptActive**, see Making an ISR Active or Inactive.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS

IoConnectInterruptEx

IoDisconnectInterruptEx

IoReportInterruptInactive

IoReportInterruptInactive function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReportInterruptInactive** routine informs the operating system that a registered interrupt service routine (ISR) is inactive and is not expecting interrupt requests.

Syntax

```
void IoReportInterruptInactive(
  PIO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS Parameters
);
```

Parameters

`Parameters`

A pointer to an IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS structure that contains the connection context associated with the interrupt. The caller received this context from the IoConnectInterruptEx call that registered the ISR.

Return Value

None.

Remarks

The IoConnectInterruptEx routine registers an ISR and connects the ISR to an interrupt or interrupts. After the ISR is registered, the driver can make the ISR active or inactive by calling the IoReportInterruptActive or **IoReportInterruptInactive** routine. By default, the ISR is active after the **IoConnectInterruptEx** call.

An ISR that is in the inactive state can be disconnected or made active. To disconnect the ISR and delete its registration, call the IoDisconnectInterruptEx routine. To make the ISR active without changing its registration, call **IoReportInterruptActive**.

The **IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS** structure must contain a valid connection contect obtained from an **IoConnectInterruptEx** call. Otherwise, the **IoReportInterruptInactive** routine bug checks in a checked build.

For more information about **IoReportInterruptInactive**, see Making an ISR Active or Inactive.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IO_REPORT_INTERRUPT_ACTIVE_STATE_PARAMETERS

IoConnectInterruptEx

IoDisconnectInterruptEx

IoReportInterruptActive

IoReportTargetDeviceChange function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReportTargetDeviceChange** routine notifies the PnP manager that a custom event has occurred on a device.

Syntax

```
NTSTATUS IoReportTargetDeviceChange(
  PDEVICE_OBJECT PhysicalDeviceObject,
  PVOID          NotificationStructure
);
```

Parameters

`PhysicalDeviceObject`

Pointer to the PDO of the device being reported.

`NotificationStructure`

Pointer to a caller-supplied TARGET_DEVICE_CUSTOM_NOTIFICATION structure describing the custom event. The PnP manager sends this structure to drivers that registered for notification of the event.

NotificationStructure.**FileObject** must be **NULL**. *NotificationStructure*.**Event** must contain the custom GUID for the event. The other fields of the *NotificationStructure* must be filled in as appropriate for the custom event.

The PnP manager fills in the *NotificationStructure*.**FileObject** field when it sends notifications to registrants.

Return Value

IoReportTargetDeviceChange returns STATUS_SUCCESS or an appropriate error status. Possible error status values include the following.

RETURN CODE	DESCRIPTION
STATUS_INVALID_DEVICE_REQUEST	The caller specified a system PnP event, such as GUID_TARGET_DEVICE_QUERY_REMOVE. This routine is only for custom events.

Remarks

After **IoReportTargetDeviceChange** notifies the PnP manager that a custom event has occurred on a device, the PnP manager sends notification of the event to drivers that registered for notification on the device. Do not use this

routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

A driver that defines a custom device event calls **IoReportTargetDeviceChange** to inform the PnP manager that the custom event has occurred. Custom notification can be used for events like a volume label change.

A driver should call the asynchronous form of this routine, IoReportTargetDeviceChangeAsynchronous, instead of this routine, to prevent deadlocks.

Certain kernel-mode components can call this synchronous routine. For example, a file system can call **IoReportTargetDeviceChange** to report a "get off the volume" custom event when a component tries to open the volume for exclusive access. Clients that register for notification on file system volumes are careful to not request an exclusive open in a PnP notification callback routine.

The custom notification structure contains a driver-defined event with its own GUID. Driver writers can generate GUIDs with Uuidgen.exe or Guidgen.exe (which are included in the Microsoft Windows SDK).

Callers of **IoReportTargetDeviceChange** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread. To report a target device change from IRQL > PASSIVE_LEVEL, call **IoReportTargetDeviceChangeAsynchronous**.

IoReportTargetDeviceChange is not supported on Windows 98/Me; it returns STATUS_NOT_IMPLEMENTED.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoReportTargetDeviceChangeAsynchronous

TARGET_DEVICE_CUSTOM_NOTIFICATION

IoReportTargetDeviceChangeAsynchronous function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReportTargetDeviceChangeAsynchronous** routine notifies the PnP manager that a custom event has occurred on a device.

Syntax

```
NTSTATUS IoReportTargetDeviceChangeAsynchronous(
  PDEVICE_OBJECT                  PhysicalDeviceObject,
  PVOID                           NotificationStructure,
  PDEVICE_CHANGE_COMPLETE_CALLBACK Callback,
  PVOID                           Context
);
```

Parameters

`PhysicalDeviceObject`

Pointer to the PDO of the device being reported.

`NotificationStructure`

Pointer to a caller-supplied TARGET_DEVICE_CUSTOM_NOTIFICATION structure describing the custom event. The PnP manager sends this structure to drivers that registered for notification of the event.

NotificationStructure.**FileObject** must be **NULL**. *NotificationStructure*.**Event** must contain the custom GUID for the event. The other fields of the *NotificationStructure* must be filled in as appropriate for the custom event.

The PnP manager fills in the *NotificationStructure*.**FileObject** field when it sends notifications to registrants.

`Callback`

Optionally points to a caller-supplied routine that the PnP manager calls after it finishes notifying drivers that registered for this custom event.

The callback routine has the following type:

```
typedef
VOID
(*PDEVICE_CHANGE_COMPLETE_CALLBACK)(
    IN PVOID Context
    );
```

A device-change-complete callback routine should not block and must not call synchronous routines that generate PnP events.

The PnP manager calls device-change-complete callback routines at IRQL = PASSIVE_LEVEL.

`Context`

Optionally points to a caller-supplied context structure that the PnP manager passes to the *Callback* routine. The caller must allocate this structure from nonpaged memory.

Return Value

IoReportTargetDeviceChangeAsynchronous returns STATUS_SUCCESS or an appropriate error status. Possible error status values include the following.

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ DEV ICE_ REQ UES T	The caller specified a system PnP event, such as GUID_TARGET_DEVICE_QUERY_REMOVE. This routine is only for custom events.

Remarks

After the **IoReportTargetDeviceChangeAsynchronous** routine notifies the PnP manager that a custom event has occurred on a device, the routine returns immediately; it does not wait while the PnP manager sends notification of the event to drivers that registered for notification on the device. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

A driver that defines a custom device event calls **IoReportTargetDeviceChangeAsynchronous** to inform the PnP manager that the custom event has occurred. Custom notification can be used for events like a volume label change.

The custom notification structure contains a driver-defined event with its own GUID. Driver writers can generate GUIDs with Uuidgen.exe or Guidgen.exe (which are included in the Microsoft Windows SDK).

When a driver calls this routine while handling an event, an IRP_MN_REMOVE_DEVICE, or an IRP_MN_SURPRISE_REMOVAL, the PnP manager calls the driver's *Callback* routine after the driver returns and the stack unwinds.

Callers of **IoReportTargetDeviceChangeAsynchronous** must be running at IRQL <= DISPATCH_LEVEL. If a driver writer calls this routine at IRQL = DISPATCH_LEVEL, the *NotificationStructure* must be allocated from nonpaged memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoReportTargetDeviceChange

TARGET_DEVICE_CUSTOM_NOTIFICATION

IoRequestDeviceEject function

1/25/2019 • 3 minutes to read • Edit Online

The **IoRequestDeviceEject** routine notifies the PnP manager that the device eject button was pressed.

Syntax

```
void IoRequestDeviceEject(
  PDEVICE_OBJECT PhysicalDeviceObject
);
```

Parameters

`PhysicalDeviceObject`

Pointer to the PDO for the device.

Return Value

None

Remarks

Note that this routine reports a request for device eject, not media eject.

Typically, a PnP bus driver calls **IoRequestDeviceEject** to notify the PnP manager that a user pressed the device eject button on one of its child devices.

A driver calls this routine, rather than sending an IRP_MN_EJECT request, because this routine allows the PnP manager to coordinate additional actions for the eject besides sending the IRP. For example, the PnP manager notifies user-mode and kernel-mode components that registered for notification of changes on the device.

The PnP manager directs an orderly shutdown of the device. The PnP manager:

1. Creates a list of other devices that are affected by this device being ejected.

   The PnP manager queries for the device's removal relations, ejection relations, and bus relations (child devices).

2. Determines whether the device and its related devices can be software-removed.

   The PnP manager sends IRP_MN_QUERY_REMOVE_DEVICE IRPs to the drivers for the device and its related devices. The PnP manager also sends notifications to any user-mode and kernel-mode components that registered for device-change notification on the device or any of its related devices. If any of the drivers or user-mode components fail the query-remove, the PnP manager pops up a dialog box to notify the user that the eject failed.

3. Software-removes the device and its related devices.

   If the previous steps are successful, the PnP manager notifies registered drivers and applications that the device and its related devices are being software-removed. Then the PnP manager sends IRP_MN_REMOVE_DEVICE IRPs to the drivers for the device and its related devices. Function and filter drivers detach from the device stack and delete their device objects for the device(s). The bus drivers retain

the PDO(s) for the device(s), unless a device is physically gone and the bus driver omitted the device in its most recent response to IRP_MN_QUERY_DEVICE_RELATIONS/**BusRelations** for the device's parent bus.

4. Directs the bus driver to eject the device (if possible).

   The PnP manager takes different steps, depending on the eject capabilities of the device:

   - Hot eject is supported.

     If the **EjectSupported** capability is set for the device, the device can be ejected when the system is running (is in the **PowerSystemWorking** state). The PnP manager sends an IRP_MN_EJECT request to the bus driver for the device. Any function and filter drivers detached previously from the stack in response to the remove IRP, so the bus driver handles the eject IRP. When the bus driver completes the IRP, the PnP manager expects the device to be physically absent from the system.

   - Hot eject is not supported.

     In this case, the device is **Removable** but does not support eject. The PnP manager marks the device as not-present/not-working-properly. The PnP manager will not restart the device until a user physically removes it and reinserts it. In this case, the PnP manager does not send an **IRP_MN_EJECT**.

   A device's parent bus driver sets the capabilities for a device, including its eject capabilities, in response to an IRP_MN_QUERY_CAPABILITIES request. A function or filter driver can optionally specify capabilities.

When a device is ejected, its child devices are physically removed from the system along with it.

A user-mode application can initiate a device eject. In that case, no driver calls this routine but the operating system calls the PnP manager to initiate the steps listed above.

Callers of **IoRequestDeviceEject** must be running at IRQL <= DISPATCH_LEVEL. The PnP manager performs most of its device-eject tasks listed above at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

IRP_MN_EJECT

IRP_MN_QUERY_DEVICE_RELATIONS

IRP_MN_QUERY_REMOVE_DEVICE

IoRequestDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **IoRequestDpc** routine queues a driver-supplied DpcForIsr routine to complete interrupt-driven I/O processing at a lower IRQL.

Syntax

```
void IoRequestDpc(
  PDEVICE_OBJECT         DeviceObject,
  PIRP                   Irp,
  __drv_aliasesMem PVOID Context
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the request that caused the interrupt is being processed.

`Irp`

Pointer to the current IRP for the specified device.

`Context`

Pointer to a driver-determined context to be passed to the DPC routine.

Return Value

None

Remarks

Callers of **IoRequestDpc** must be running at DIRQL.

Drivers call **IoRequestDpc** from an InterruptService routine. Because of this, **IoRequestDpc** runs at the DIRQL value that was specified by *SynchronizeIrql* when the driver called IoConnectInterrupt. However, it is also possible to queue a DPC at any IRQL >= DISPATCH_LEVEL by using the **Ke*Xxx*Dpc** routines. For more information, see Which Type of DPC Should You Use?

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

IRQL	DIRQL

See Also

IoInitializeDpcRequest

KeInitializeDpc

KeInsertQueueDpc

IoReuseIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **IoReuseIrp** routine reinitializes an IRP so that it can be reused.

Syntax

```
void IoReuseIrp(
  PIRP     Irp,
  NTSTATUS Iostatus
);
```

Parameters

`Irp`

Pointer to the IRP to be reinitialized for reuse.

`Iostatus`

Return Value

None

Remarks

Drivers for Windows 2000 and later versions of Windows use **IoReuseIrp** to reuse an IRP.

A driver should use **IoReuseIrp** only on IRPs it previously allocated either as raw memory or with IoAllocateIrp. In particular, drivers should not use this routine for IRPs created with IoMakeAssociatedIrp, IoBuildSynchronousFsdRequest, IoBuildAsynchronousFsdRequest, or IoBuildDeviceIoControlRequest.

See Reusing IRPs for more details on how to reuse IRPs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

DDI compliance rules	IoReuseIrp, IoReuseIrp2, HwStorPortProhibitedDDIs

See Also

IRP

IoAllocateIrp

IoInitializeIrp

IoMakeAssociatedIrp

IoSetCancelRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetCancelRoutine** routine sets up a driver-supplied Cancel routine to be called if a given IRP is canceled.

Syntax

```
PDRIVER_CANCEL IoSetCancelRoutine(
  PIRP           Irp,
  PDRIVER_CANCEL CancelRoutine
);
```

Parameters

`Irp`

Pointer to the IRP being put into or removed from a cancelable state.

`CancelRoutine`

Specifies the entry point of the caller-supplied Cancel routine to be called if the specified IRP is canceled or is **NULL** if the given IRP is being removed from the cancelable state. This routine is declared as follows:

```
VOID
(*PDRIVER_CANCEL)(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

Return Value

IoSetCancelRoutine returns the previous value of *Irp*->**CancelRoutine**. If no *Cancel* routine was previously set, or if IRP cancellation is already in progress, **IoSetCancelRoutine** returns **NULL**.

Remarks

This routine can disable the *Cancel* routine currently set in an IRP.

A driver must hold the system cancel spin lock when calling this routine if the driver uses the I/O manager-supplied device queue in the device object. The driver runs at IRQL = DISPATCH_LEVEL after calling IoAcquireCancelSpinLock until it releases the cancel spin lock with IoReleaseCancelSpinLock.

If the driver manages its own queues of IRPs, then the driver need not hold the cancel spin lock when calling this routine. **IoSetCancelRoutine** uses an interlocked exchange intrinsic to set the address of the *Cancel* routine as an atomic operation. Reduced usage of the cancel spin lock can improve driver performance and overall system performance.

Driver *Cancel* routines are called at IRQL = DISPATCH_LEVEL with the cancel spin lock held. The *Cancel* routine must release the cancel spin lock before it returns control.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL (see Remarks section)
DDI compliance rules	IrpCancelField, StartIoCancel

See Also

IoAcquireCancelSpinLock

IoReleaseCancelSpinLock

IoSetCompletionRoutine function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetCompletionRoutine** routine registers an IoCompletion routine, which will be called when the next-lower-level driver has completed the requested operation for the given IRP.

Syntax

```
void IoSetCompletionRoutine(
  PIRP                    Irp,
  PIO_COMPLETION_ROUTINE CompletionRoutine,
  __drv_aliasesMem PVOID Context,
  BOOLEAN                 InvokeOnSuccess,
  BOOLEAN                 InvokeOnError,
  BOOLEAN                 InvokeOnCancel
);
```

Parameters

`Irp`

Pointer to the IRP that the driver is processing.

`CompletionRoutine`

Specifies the entry point for the driver-supplied IoCompletion routine, which is called when the next-lower driver completes the packet.

`Context`

Pointer to a driver-determined context to pass to the IoCompletion routine. Context information must be stored in nonpaged memory, because the *IoCompletion* routine is called at IRQL <= DISPATCH_LEVEL.

`InvokeOnSuccess`

Specifies whether the completion routine is called if the IRP is completed with a success status value in the IRP's IO_STATUS_BLOCK structure, based on results of the **NT_SUCCESS** macro (see Using NTSTATUS values).

`InvokeOnError`

Specifies whether the completion routine is called if the IRP is completed with a nonsuccess status value in the IRP's **IO_STATUS_BLOCK** structure.

`InvokeOnCancel`

Specifies whether the completion routine is called if a driver or the kernel has called IoCancelIrp to cancel the IRP.

Return Value

None

Remarks

Note Only a driver that can guarantee it will not be unloaded before its completion routine finishes can use **IoSetCompletionRoutine**. Otherwise, the driver must use IoSetCompletionRoutineEx, which prevents the driver from unloading until its completion routine executes.

This routine sets the transfer address of the IoCompletion routine in the given IRP. The lowest-level driver in a chain of layered drivers cannot call this routine.

IoSetCompletionRoutine registers the specified routine to be called when the next-lower-level driver has completed the requested operation in any or all of the following ways:

- With a success status value
- With a nonsuccess status value
- By canceling the IRP

Usually, the I/O status block is set by the underlying device driver. It is read but not altered by any higher-level drivers' *IoCompletion* routines.

Higher-level drivers that allocate IRP's with IoAllocateIrp or IoBuildAsynchronousFsdRequest must call this routine with all *InvokeOn*Xxx parameters set to **TRUE** before passing the driver-allocated IRP to IoCallDriver. When the *IoCompletion* routine is called with such an IRP, it must free the driver-allocated IRP and any other resources that the driver set up for the request, such as MDLs with IoBuildPartialMdl. Such a driver should return STATUS_MORE_PROCESSING_REQUIRED when it calls IoFreeIrp to forestall the I/O manager's completion processing for the driver-allocated IRP.

Non-PnP drivers that might be unloaded before their *IoCompletion* routines run should use IoSetCompletionRoutineEx instead.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	CompleteRequest, CompleteRequestStatusCheck, CompletionRoutineRegistered, IoAllocateForward, IoAllocateIrpSignalEventInCompletion, IoAllocateIrpSignalEventInCompletion2, IoAllocateIrpSignalEventInCompletion3, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildFsdForward, IoBuildFsdIrpSignalEventInCompletion, IoBuildFsdIrpSignalEventInCompletion2, IoBuildFsdIrpSignalEventInCompletion3, IoBuildFsdIrpSignalEventInCompletionTimeout, IoSetCompletionRoutineNonPnpDriver, LowerDriverReturn, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequest, SignalEventInCompletion, SignalEventInCompletion2, SignalEventInCompletion3, StartDeviceWait, StartDeviceWait3, SetCompletionRoutineFromDispatch, IoFreeIrp

See Also

IO_STACK_LOCATION

IRP

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildPartialMdl

IoFreeIrp

IoSetCompletionRoutineEx

IoSetCompletionRoutineEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetCompletionRoutineEx** routine registers an IoCompletion routine, which is called when the next-lower-level driver has completed the requested operation for the given IRP.

Syntax

```
NTSTATUS IoSetCompletionRoutineEx(
  PDEVICE_OBJECT         DeviceObject,
  PIRP                   Irp,
  PIO_COMPLETION_ROUTINE CompletionRoutine,
  PVOID                  Context,
  BOOLEAN                InvokeOnSuccess,
  BOOLEAN                InvokeOnError,
  BOOLEAN                InvokeOnCancel
);
```

Parameters

`DeviceObject`

Pointer to the driver's device object.

`Irp`

Pointer to the IRP that the driver is processing.

`CompletionRoutine`

Specifies the entry point for the driver-supplied IoCompletion routine, which is called when the next-lower driver completes the packet.

`Context`

Pointer to a driver-determined context to pass to the *IoCompletion* routine. Context information must be stored in nonpaged memory, because the *IoCompletion* routine is called at IRQL <= DISPATCH_LEVEL.

`InvokeOnSuccess`

Specifies whether the completion routine is called if the IRP is completed with a success status value in the IRP's IO_STATUS_BLOCK structure, based on results of the NT_SUCCESS macro (see Using NTSTATUS values).

`InvokeOnError`

Specifies whether the completion routine is called if the IRP is completed with a nonsuccess status value in the IRP's **IO_STATUS_BLOCK** structure.

`InvokeOnCancel`

Specifies whether the completion routine is called if a driver or the kernel has called IoCancelIrp to cancel the IRP.

Return Value

This routine returns STATUS_SUCCESS on success, or STATUS_INSUFFICIENT_RESOURCES if insufficient

memory is available for the operation.

Remarks

> **Note** Unlike IoSetCompletionRoutine, the **IoSetCompletionRoutineEx** routine returns an NTSTATUS value. The driver must check this value to determine if the IoCompletion routine was successfully registered. If the *IoCompletion* routine is successfully registered, **IoSetCompletionRoutineEx** allocates memory that remains allocated until the *IoCompletion* routine executes. Drivers must ensure that their *IoCompletion* routine executes by calling IoCallDriver; otherwise, the kernel will leak memory.

The *IoCompletion* routine must belong to the driver that owns the device object pointed to by *DeviceObject*. This requirement prevents the *IoCompletion* routine from being unloaded before it returns.

The behavior of **IoSetCompletionRoutineEx** is the same as the IoSetCompletionRoutine routine, except that:

- **IoSetCompletionRoutineEx** guarantees that a non-Plug and Play driver is not unloaded before the *IoCompletion* routine runs.
- **IoSetCompletionRoutineEx** is a routine that returns an NTSTATUS value.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	CompleteRequest, CompleteRequestStatusCheck, CompletionRoutineRegistered, IoAllocateForward, IoAllocateIrpSignalEventInCompletion, IoAllocateIrpSignalEventInCompletion2, IoAllocateIrpSignalEventInCompletion3, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildFsdForward, IoBuildFsdIrpSignalEventInCompletion, IoBuildFsdIrpSignalEventInCompletion2, IoBuildFsdIrpSignalEventInCompletion3, IoBuildFsdIrpSignalEventInCompletionTimeout, IoSetCompletionExCompleteIrp, IoSetCompletionRoutineExCheck, IoSetCompletionRoutineExCheckDeviceObject, LowerDriverReturn, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequestEx, SignalEventInCompletion, SignalEventInCompletion2, SignalEventInCompletion3, StartDeviceWait2, StartDeviceWait4, SetCompletionRoutineFromDispatch, HwStorPortProhibitedDDIs

See Also

IO_STACK_LOCATION

IRP

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildPartialMdl

IoCompletion

IoFreeIrp

IoSetCompletionRoutine

IoSetDeviceInterfacePropertyData function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetDeviceInterfacePropertyData** routine modifies the current value of a device interface property.

Syntax

```
NTSTATUS IoSetDeviceInterfacePropertyData(
  PUNICODE_STRING  SymbolicLinkName,
  const DEVPROPKEY *PropertyKey,
  LCID             Lcid,
  ULONG            Flags,
  DEVPROPTYPE      Type,
  ULONG            Size,
  PVOID            Data
);
```

Parameters

`SymbolicLinkName`

A pointer to a string that identifies the device interface instance. This string was obtained from a previous call to the IoGetDeviceInterfaces, IoGetDeviceInterfaceAlias, or IoRegisterDeviceInterface routine.

`PropertyKey`

A pointer to a DEVPROPKEY structure that contains the device interface property key.

`Lcid`

Specifies a locale identifier. Set this parameter either to a language-specific LCID value or to **LOCALE_NEUTRAL**. The **LOCALE_NEUTRAL** LCID specifies that the property is language-neutral (that is, not specific to any language). Do not set this parameter to **LOCALE_SYSTEM_DEFAULT** or **LOCALE_USER_DEFAULT**. For more information about language-specific LCID values, see LCID Structure.

`Flags`

Set this parameter to **PLUGPLAY_PROPERTY_PERSISTENT** if the property value set by this routine should persist across computer restarts. Otherwise, set *Flags* to zero.

`Type`

Set this parameter to the DEVPROPTYPE value that specifies the type of the data that is supplied in the *Data* buffer.

`Size`

Specifies the size, in bytes, of the buffer that *Data* points to.

`Data`

A pointer to the device interface property data. Set this parameter to **NULL** to delete the specified property. If *Data* is non-**NULL**, the routine stores an internal copy of the property value. The buffer pointed to by *Data* does not need to remain valid after the call returns.

Return Value

IoSetDeviceInterfacePropertyData returns STATUS_SUCCESS if the call was successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STA TUS _UN SUC CES SFU L	The specified LCID value is not valid.
STA TUS _NO T_I MP LEM ENT ED	The specified property is not supported.

Remarks

Kernel-mode drivers use the **IoSetDeviceInterfacePropertyData** routine to modify device interface properties that are defined as part of the unified device property model. For more information about device interface properties, see Device Properties.

Drivers can use the IoGetDeviceInterfacePropertyData routine to obtain the current value for a device interface property.

Callers of **IoSetDeviceInterfacePropertyData** must be running at IRQL <= APC_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available for Windows 8 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DEVPROPKEY

DEVPROPTYPE

IoGetDeviceInterfacePropertyData

IoSetDeviceInterfaceState function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetDeviceInterfaceState** routine enables or disables an instance of a previously registered device interface class.

Syntax

```
NTSTATUS IoSetDeviceInterfaceState(
  PUNICODE_STRING SymbolicLinkName,
  BOOLEAN         Enable
);
```

Parameters

`SymbolicLinkName`

Pointer to a string that identifies the device interface instance that is being enabled or disabled. This string was obtained from a previous call to IoRegisterDeviceInterface or IoGetDeviceInterfaces.

`Enable`

TRUE indicates that the device interface is being enabled. **FALSE** indicates that the device interface is being disabled.

Return Value

IoSetDeviceInterfaceState returns STATUS_SUCCESS if the call was successful. This routine returns an informational status of STATUS_OBJECT_NAME_EXISTS if the caller requested to enable a device interface that was already enabled. Possible error return values are described following.

RETURN CODE	DESCRIPTION
STATUS_OBJECT_NAME_NOT_FOUND	The caller tried to disable a device interface that was not enabled.

Remarks

IoSetDeviceInterfaceState enables an instance of a registered device interface for use by applications and other system components. The interface class must have been previously registered with IoRegisterDeviceInterface or from user mode.

Applications and other system components can open only interfaces that are enabled.

A function or a filter driver typically calls this routine with *Enable* set to **TRUE** after it successfully starts a device in response to an IRP_MN_START_DEVICE IRP. Such a driver should disable the device interface instance (that is, call **IoSetDeviceInterfaceState** and set *Enable* to **FALSE**) when it removes the device in response to an IRP_MN_REMOVE_DEVICE IRP or an IRP_MN_SURPRISE_REMOVAL IRP. If a driver does not disable a device interface when it processes these removal IRPs, the driver should not subsequently attempt to do so because the PnP manager will disable the interface when the PnP manager removes the device.

If a device is removed suddenly (for example, by a surprise removal) but still has a valid device interface instance, a problem will occur if the device is reattached. This problem occurs when the PnP manager enumerates the newly attached device and enables a device interface instance, which will exist at the same registry path as the existing device interface instance.

Note that if the driver calls **IoSetDeviceInterfaceState** to disable a device interface instance in response to an **IRP_MN_SURPRISE_REMOVAL** IRP, the driver must not attempt to disable the same device interface instance in response to an **IRP_MN_REMOVE_DEVICE** IRP.

If a call to **IoSetDeviceInterfaceState** successfully exposes a device interface instance, the system notifies any components that registered for PnP notification of a device class change. Similarly, if a call to this routine disables an existing device interface instance, the system sends appropriate notifications.

The PnP manager does not send notification of interface instance arrivals until the **IRP_MN_START_DEVICE** IRP completes, indicating that all the drivers for the device have completed their start operations. In addition, the PnP manager fails create requests for the device until the **IRP_MN_START_DEVICE** IRP completes.

Callers of **IoSetDeviceInterfaceState** must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	IrqlIoPassive1, LowerDriverReturn, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IRP_MN_REMOVE_DEVICE

IRP_MN_START_DEVICE

IRP_MN_SURPRISE_REMOVAL

IoGetDeviceInterfaces

IoRegisterDeviceInterface

IoRegisterPlugPlayNotification

IoSetDevicePropertyData function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetDevicePropertyData** routine modifies the current setting for a device property.

Syntax

```
NTSTATUS IoSetDevicePropertyData(
  PDEVICE_OBJECT   Pdo,
  const DEVPROPKEY *PropertyKey,
  LCID             Lcid,
  ULONG            Flags,
  DEVPROPTYPE      Type,
  ULONG            Size,
  PVOID            Data
);
```

Parameters

`Pdo`

A pointer to the physical device object (PDO) for the device that is being queried.

`PropertyKey`

A pointer to a DEVPROPKEY structure that specifies the device property key.

`Lcid`

A locale identifier. Set this parameter either to a language-specific LCID value or to **LOCALE_NEUTRAL**.

The **LOCALE_NEUTRAL** LCID specifies that the property is language-neutral (that is, not specific to any language).

Do not set this parameter to **LOCALE_SYSTEM_DEFAULT** or **LOCALE_USER_DEFAULT**.

For more information about language-specific LCID values, see LCID Structure.

`Flags`

Set this parameter to **PLUGPLAY_PROPERTY_PERSISTENT** if the property value set by this routine should persist across computer restarts. Otherwise, set *Flags* to zero.

Windows 8 and Windows Server 2012 and later operating systems treat **Flags** as if **PLUGPLAY_PROPERTY_PERSISTENT** is always passed.

`Type`

A DEVPROPTYPE value that specifies the type of the data that is provided in the *Data* buffer.

`Size`

The size, in bytes, of the buffer that *Data* points to.

`Data`

A pointer to the device property data. Set this parameter to **NULL** to delete the specified property. If *Data* is non-**NULL**, the routine stores an internal copy of the property value. The buffer pointed to by *Data* does not need to remain valid after the call returns.

Return Value

IoSetDevicePropertyData returns STATUS_SUCCESS if the call was successful, or the appropriate NTSTATUS code on failure.

Remarks

Kernel-mode drivers use the **IoSetDevicePropertyData** routine to modify device properties that are defined as part of the unified device property model. For more information about device properties, see Device Properties.

To delete a property for a specific locale, pass a language-specific LCID value in *Lcid* and **NULL** in *Data*.

To delete a property for all locales, pass **LOCALE_NEUTRAL** in *Lcid* and **NULL** in *Data*.

To modify a property for a specific locale, pass a language-specific LCID value in *Lcid* and non-**NULL** in *Data*.

To modify a property for all locales, pass **LOCALE_NEUTRAL** in *Lcid* and non-**NULL** in *Data*.

Beginning with Windows 8 and Windows Server 2012 passing **LOCALE_NEUTRAL** in *Lcid* is treated like any other locale.

Drivers can use the IoGetDevicePropertyData routine to obtain the current value for a device property.

Callers of **IoSetDeviceProperty** must be running at IRQL <= APC_LEVEL in the context of a system thread.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

DEVPROPKEY

DEVPROPTYPE

IoGetDevicePropertyData

IoSetIoPriorityHint function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetIoPriorityHint** routine sets the priority hint value for an IRP.

Syntax

```
NTSTATUS IoSetIoPriorityHint(
  PIRP             Irp,
  IO_PRIORITY_HINT PriorityHint
);
```

Parameters

`Irp`

Specifies the IRP to set the priority hint value for.

`PriorityHint`

Specifies the IO_PRIORITY_HINT value that indicates the new priority hint.

Return Value

IoSetIoPriorityHint returns STATUS_SUCCESS if the operation succeeds and the appropriate NTSTATUS value if the operation fails.

Remarks

For more information about priority hints, see Using IRP Priority Hints.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_PRIORITY_HINT

IoSetLinkShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetLinkShareAccess** routine sets the access rights for link sharing the specified file object.

Syntax

```
void IoSetLinkShareAccess(
  ACCESS_MASK        DesiredAccess,
  ULONG              DesiredShareAccess,
  PFILE_OBJECT       FileObject,
  PSHARE_ACCESS      ShareAccess,
  PLINK_SHARE_ACCESS LinkShareAccess,
  ULONG              IoShareAccessFlags
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that indicates the desired type of access to the given file object.

`DesiredShareAccess`

Specifies the desired type of shared access to the file object for the current open request. The value of this parameter is usually the same as the *ShareAccess* parameter that is passed to the file system or highest-level driver by the I/O manager when the open request was made. This value can be zero, or any combination of the following:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

A pointer to the file object for which to check access for the current open request.

`ShareAccess`

A pointer to the common share-access data structure that is associated with *FileObject*. Drivers should treat this structure as opaque.

`LinkShareAccess`

A pointer to the common link share-access data structure (LINK_SHARE_ACCESS) that is associated with *FileObject*. Drivers should treat this structure as opaque.

`IoShareAccessFlags`

A bitmask of these flags:

IO_SHARE_ACCESS_NO_WRITE_PERMISSION (0x80000000) specifies that the user has no write permission for the file. This flag is used to prevent opening a file for exclusive read access when the user does not have appropriate permissions.

IO_CHECK_SHARE_ACCESS_UPDATE_SHARE_ACCESS (0x00000001) indicates whether the SHARE_ACCESS structure is updated.

Return Value

This function does not return a value.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib

See Also

IoCheckLinkShareAccess

IoRemoveLinkShareAccess

IoUpdateLinkShareAccess

IoSetNextIrpStackLocation function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetNextIrpStackLocation** routine sets the IRP stack location in a driver-allocated IRP to that of the caller.

Syntax

```
void IoSetNextIrpStackLocation(
  PIRP Irp
);
```

Parameters

`Irp`

Pointer to the IRP whose stack location is to be set.

Return Value

None

Remarks

In general, this routine is seldom used by drivers. It is primarily used by drivers that require their own stack location in an IRP that they have allocated, on their own, to send to another driver.

IoSetNextIrpStackLocation is generally not needed because either:

- The driver received the IRP it is passing from another, higher-level driver, and so it already owns a stack location,
- Or, the driver allocated the IRP but does not need its own stack location because it can keep everything that it needs in a context block whose address can be passed to its *IoCompletion* routine.

Care should be taken if this routine is called, especially when allocating the IRP with **IoAllocateIrp** or **IoMakeAssociatedIrp**. The writer of the allocating driver must remember that a caller-specific stack location is not included in the number of stack locations required by the lower-level drivers to which it sends IRPs with **IoCallDriver**. A driver must explicitly specify a stack location for itself in its call to **IoAllocateIrp** or **IoMakeAssociatedIrp** if it calls **IoSetNextIrpStackLocation** with the IRP returned by either routine.

A driver cannot call **IoSetNextIrpStackLocation** with any IRP it allocates by calling **IoBuildAsynchronousFsdRequest**, **IoBuildDeviceIoControlRequest**, or **IoBuildSynchronousFsdRequest**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IO_STACK_LOCATION

IoAllocateIrp

IoBuildAsynchronousFsdRequest

IoBuildDeviceIoControlRequest

IoBuildSynchronousFsdRequest

IoCallDriver

IoSetCompletionRoutine

IoSetShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetShareAccess** routine sets the access rights for sharing the given file object.

Syntax

```
void IoSetShareAccess(
  ACCESS_MASK   DesiredAccess,
  ULONG         DesiredShareAccess,
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that represents the type of access requested for the *FileObject*. See IoCreateFile for a complete list of system-defined *DesiredAccess*flags.

`DesiredShareAccess`

Specifies the type of share access to be set for the file object. This value can be zero, or any combination of the following:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

Pointer to the file object whose share access is being set or reset.

`ShareAccess`

Pointer to the SHARE_ACCESS structure associated with *FileObject*. Drivers should treat this structure as opaque.

Return Value

None

Remarks

Only highest-level kernel-mode drivers should call this routine. The call must occur in the context of the first thread that attempts to open the *FileObject*.

This routine sets the access and share access information when the *FileObject* is first opened. It returns a pointer to the common share-access data structure associated with *FileObject*. Callers should save this pointer for later use when updating the access or closing the file.

Generally, file system drivers (FSDs) are most likely to call this routine. However, other highest-level drivers can

call **IoSetShareAccess** to control the kind of access allowed to a driver-created device object associated with the given *FileObject*.

IoSetShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoSetShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

IoCheckShareAccess

IoCreateFile

IoCreateFileEx

IoGetFileObjectGenericMapping

IoGetRelatedDeviceObject

IoRemoveShareAccess

IoUpdateShareAccess

IoSetShareAccessEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetShareAccessEx** routine sets the access rights for sharing the specified file object.

Syntax

```
void IoSetShareAccessEx(
  ACCESS_MASK   DesiredAccess,
  ULONG         DesiredShareAccess,
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess,
  PBOOLEAN      WritePermission
);
```

Parameters

`DesiredAccess`

Specifies an ACCESS_MASK value that represents the type of access requested for the file object. See IoCreateFile for a complete list of system-defined *DesiredAccess* flags.

`DesiredShareAccess`

Specifies the type of share access to be set for the file object. This value can be zero, or any combination of the following flags:

FILE_SHARE_READ

FILE_SHARE_WRITE

FILE_SHARE_DELETE

`FileObject`

A pointer to the file object whose share access is being set or reset.

`ShareAccess`

A pointer to the **SHARE_ACCESS** structure that is associated with *FileObject*. Drivers should treat this structure as opaque.

`WritePermission`

A pointer to the value that specifies whether the file object has write permission. This value is **TRUE** if the share has write permission; otherwise, it is **FALSE**. If the value is **FALSE** and the caller attempts to take exclusive read access, the write permission is downgraded to FILE_SHARE_READ.

Return Value

None.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib

See Also

ACCESS_MASK

IoCreateFile

IoSetStartIoAttributes function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSetStartIoAttributes** routine sets attributes for the driver's StartIo routine.

Syntax

```
void IoSetStartIoAttributes(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        DeferredStartIo,
  BOOLEAN        NonCancelable
);
```

Parameters

`DeviceObject`

Pointer to the device object for the driver's device.

`DeferredStartIo`

If **TRUE**, the I/O manager will defer any call to the driver's StartIo routine while the driver is already inside the routine. In particular, if the *StartIo* routine calls IoStartNextPacket, the *StartIo* routine will not be called again until the current invocation completes. The default is **FALSE**.

`NonCancelable`

If **TRUE**, the IRP cannot be canceled once it has been dequeued by a call to **IoStartNextPacket**. The default is **FALSE**. Drivers that set this member to **FALSE** must synchronize their IRP handling with the cancel spin lock.

Return Value

None

Requirements

Minimum supported client	Available for Microsoft Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

DDI compliance rules	StartIoCancel, StartIoRecursion, HwStorPortProhibitedDDIs

See Also

IoStartNextPacket

StartIo

IoSizeOfIrp macro

1/25/2019 • 2 minutes to read • Edit Online

The **IoSizeOfIrp** routine determines the size in bytes for an IRP, given the number of stack locations in the IRP.

Syntax

```
void IoSizeOfIrp(
   StackSize
);
```

Parameters

`StackSize`

Specifies the number of stack locations for the IRP.

Return Value

IoSizeOfIrp returns the size, in bytes, of the IRP.

Remarks

The input *StackSize* value is either that of the next-lower driver's device object or one more than that value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

IoAllocateIrp

IoMakeAssociatedIrp

IoSizeofWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoSizeofWorkItem** routine returns the size, in bytes, of an IO_WORKITEM structure.

Syntax

```
ULONG IoSizeofWorkItem(

);
```

Parameters

This function has no parameters.

Return Value

IoSizeofWorkItem returns the number of bytes necessary to hold an IO_WORKITEM structure.

Remarks

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_WORKITEM

IoInitializeWorkItem

IoStartNextPacket function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartNextPacket** routine dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.

Syntax

```
void IoStartNextPacket(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        Cancelable
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the IRP is to be dequeued.

`Cancelable`

Specifies whether IRPs in the device queue can be canceled.

Return Value

None

Remarks

If there are no IRPs currently in the device queue for the target *DeviceObject*, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass **TRUE** in the *Cancelable* parameter. If *Cancelable* is **TRUE**, the I/O manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a *StartIo* routine cannot call **IoStartNextPacket**.

Drivers that call **IoStartNextPacket** from their *StartIo* routine should be aware of recursion issues. If a driver can call **IoStartNextPacket** on a large number of requests in succession from its *StartIo* routine (for example, when a device error occurs and the driver is clearing out it device queue), you should set the *DeferredStartIo* attribute for the device by using IoSetStartIoAttributes routine. This attribute ensures that the next packet will not be issued until the previous *StartIo* call returns.

Callers of **IoStartNextPacket** must be running at IRQL = DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL = DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL (see Remarks section)
DDI compliance rules	IrqlDispatch, SpinLockSafe, StartIoRecursion, HwStorPortProhibitedDDIs, IrqlDispatch(storport), SpinLockSafe(storport)

See Also

DEVICE_OBJECT

IoSetStartIoAttributes

IoStartNextPacketByKey

IoStartPacket

IoStartNextPacketByKey function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartNextPacketByKey** routine dequeues the next I/O request packet from the specified device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.

Syntax

```
void IoStartNextPacketByKey(
  PDEVICE_OBJECT DeviceObject,
  BOOLEAN        Cancelable,
  ULONG          Key
);
```

Parameters

`DeviceObject`

Pointer to the device object for which the IRP is to be dequeued.

`Cancelable`

Specifies whether IRPs in the device queue can be canceled.

`Key`

Specifies the sort key that determines which entry to remove from the queue.

Return Value

None

Remarks

If there are no IRPs currently in the device queue for the target device object, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass **TRUE** in the *Cancelable* parameter. If *Cancelable* is **TRUE**, the I/O manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a *StartIo* routine cannot call **IoStartNextPacketByKey**.

Callers of **IoStartNextPacketByKey** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL = DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DEVICE_OBJECT

IoStartNextPacket

IoStartPacket

IoStartPacket function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartPacket** routine calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.

Syntax

```
void IoStartPacket(
  PDEVICE_OBJECT DeviceObject,
  PIRP           Irp,
  PULONG         Key,
  PDRIVER_CANCEL CancelFunction
);
```

Parameters

`DeviceObject`

Pointer to the target device object for the IRP.

`Irp`

Pointer to the IRP to be processed.

`Key`

Pointer to a value that determines where to insert the packet into the device queue. If this is zero, the packet is inserted at the tail of the device queue.

`CancelFunction`

Specifies the entry point for a driver-supplied Cancel routine.

Return Value

None

Remarks

If the driver is already busy processing a request for the target device object, then the packet is queued in the device queue. Otherwise, this routine calls the driver's *StartIo* routine with the specified IRP.

If a non-NULL *CancelFunction* pointer is supplied, it is set in the IRP so the driver's *Cancel* routine is called if the IRP is canceled before its completion.

Drivers that do not have a *StartIo* routine cannot call **IoStartPacket**.

Callers of **IoStartPacket** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's Dispatch routine at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

DEVICE_OBJECT

IoMarkIrpPending

IoSetCancelRoutine

IoStartNextPacket

IoStartNextPacketByKey

IoStartTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStartTimer** routine enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.

Syntax

```
void IoStartTimer(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to a device object whose timer routine is to be called.

Return Value

None

Remarks

The driver must already have set up the IoTimer routine for the *DeviceObject* by calling **IoInitializeTimer**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Windows
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoInitializeTimer

IoStopTimer

IoTimer

KeInitializeDpc

KeInitializeTimer

KeSetTimer

IoStopTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **IoStopTimer** routine disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.

Syntax

```
void IoStopTimer(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the device object with which the IoTimer routine is associated.

Return Value

None

Remarks

The driver-supplied IoTimer routine can be reenabled with a call to **IoStartTimer**.

Do not call **IoStopTimer** from within the *IoTimer* routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Windows
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoInitializeTimer

IoStartTimer

IoUninitializeWorkItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUninitializeWorkItem** routine uninitializes a work item that was initialized by IoInitializeWorkItem.

Syntax

```
void IoUninitializeWorkItem(
  PIO_WORKITEM IoWorkItem
);
```

Parameters

`IoWorkItem`

Pointer to the IO_WORKITEM structure to uninitialize.

Return Value

None

Remarks

Only uninitialize a work item that is not currently queued. The system dequeues a work item before it runs the work item's callback routine, so **IoUninitializeWorkItem** can be called from within the WorkItem or WorkItemEx routine for the work item.

For more information about work items, see System Worker Threads.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_WORKITEM

IoInitializeWorkItem

IoUnregisterContainerNotification function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUnregisterContainerNotification** routine cancels a container notification registration that was previously created by the IoRegisterContainerNotification routine.

Syntax

```
void IoUnregisterContainerNotification(
  PVOID CallbackRegistration
);
```

Parameters

`CallbackRegistration`

A pointer to a callback registration object that was created by a previous call to **IoRegisterContainerNotification**.

Return Value

None

Remarks

After this routine returns, the *CallbackRegistration* pointer value is no longer valid.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IoRegisterContainerNotification

IoUnregisterPlugPlayNotification function

1/25/2019 • 2 minutes to read • Edit Online

This routine is obsolete in Windows 7 and later versions of Windows. For more information, see the following Remarks section.

The **IoUnregisterPlugPlayNotification** routine removes the registration of a driver's callback routine for a PnP event.

Syntax

```
NTSTATUS IoUnregisterPlugPlayNotification(
  PVOID NotificationEntry
);
```

Parameters

`NotificationEntry`

Pointer to an opaque value representing the registration to be removed. The value was returned by a previous call to IoRegisterPlugPlayNotification.

Return Value

IoUnregisterPlugPlayNotification always returns STATUS_SUCCESS if *NotificationEntry*is valid.

Remarks

In Windows 7 and later versions of Windows, this function is obsolete and is provided only to support existing drivers. Use the IoUnregisterPlugPlayNotificationEx routine instead.

> **Note** The system does not synchronize between the execution of the notification routine and **IoUnregisterPlugPlayNotification**. Therefore, the routine can be called after the **IoUnregisterPlugPlayNotification** method has returned. If necessary, a driver should implement its own mechanism to ignore any notifications after **IoUnregisterPlugPlayNotification** has been called.

IoUnregisterPlugPlayNotification removes one PnP notification registration; that is, the registration of one driver callback routine for one PnP event category.

Drivers should unregister a notification first, then free any related context buffer.

A driver cannot be unloaded until it removes all of its PnP notification registrations because there is a reference on its driver object for each active registration.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoRegisterPlugPlayNotification

IoUnregisterPlugPlayNotificationEx

IoUnregisterPlugPlayNotificationEx function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUnregisterPlugPlayNotificationEx** routine cancels the registration of a driver's callback routine for notifications of Plug and Play (PnP) events.

Syntax

```
NTSTATUS IoUnregisterPlugPlayNotificationEx(
  PVOID NotificationEntry
);
```

Parameters

`NotificationEntry`

A pointer to an opaque value that represents the registration to cancel. The caller previously obtained this value by calling the IoRegisterPlugPlayNotification routine.

Return Value

IoUnregisterPlugPlayNotificationEx returns STATUS_SUCCESS if the *NotificationEntry* parameter is valid.

Remarks

A kernel-mode driver calls this routine to remove a registration to receive PnP notifications. That is, an **IoUnregisterPlugPlayNotificationEx** call cancels the registration of a driver callback routine for one PnP event category. The driver previously obtained this registration by calling the **IoRegisterPlugPlayNotification** routine.

After an **IoUnregisterPlugPlayNotificationEx** call returns, the specified registration is canceled and no further callbacks can occur.

The IoUnregisterPlugPlayNotification routine is similar to **IoUnregisterPlugPlayNotificationEx**, except that it cannot guarantee that no further callbacks can occur after a **IoUnregisterPlugPlayNotification** call returns.

Frequently, a driver calls **IoUnregisterPlugPlayNotificationEx** from a notification callback routine. In most cases, the driver can safely delete the registration of the notification callback routine in this way. However, it is unsafe for a notification callback routine to call **IoUnregisterPlugPlayNotificationEx** to unregister itself if the following are both true:

- The driver sets the PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES flag in the *EventCategoryFlags* parameter value supplied to the IoRegisterPlugPlayNotification call that registers the notification callback routine.
- The **IoRegisterPlugPlayNotification** call that registers the notification callback routine has not yet returned.

The callback routine must not call any routine that might block the execution of the thread on which the callback routine is running. For example, if a poorly designed callback routine places a work item that calls **IoUnregisterPlugPlayNotificationEx** in the work item queue, and then waits for a worker thread to complete the work item, it would result in a deadlock of the operating system.

Drivers should cancel the registration of a notification callback routine first, and then free any context buffer that is

associated with the routine.

A driver cannot be unloaded until it removes all of its PnP notification registrations because each active registration holds a counted reference to the driver object that represents the loaded image of the driver.

For more information, see Using PnP Notification.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoRegisterPlugPlayNotification

IoUnregisterPlugPlayNotification

IoUnregisterShutdownNotification function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUnregisterShutdownNotification** routine removes a registered driver from the shutdown notification queue.

Syntax

```
void IoUnregisterShutdownNotification(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to the driver's device object.

Return Value

None

Remarks

IoUnregisterShutdownNotification can be called by a driver only if that driver previously called IoRegisterShutdownNotification or IoRegisterLastChanceShutdownNotification with the given *DeviceObject*. This routine is usually called from a driver's Unload routine.

Calling **IoUnregisterShutdownNotification** cancels all shutdown notifications that have been registered for the given *DeviceObject*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoRegisterLastChanceShutdownNotification

IoRegisterShutdownNotification

IoUpdateLinkShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUpdateLinkShareAccess** routine updates the share access for the given file object, usually when the file is being opened.

Syntax

```
void IoUpdateLinkShareAccess(
  PFILE_OBJECT       FileObject,
  PSHARE_ACCESS      ShareAccess,
  PLINK_SHARE_ACCESS LinkShareAccess
);
```

Parameters

`FileObject`

Pointer to the file object, which usually is being closed by the current thread.

`ShareAccess`

Pointer to the share-access structure that describes how the open file object is currently being accessed.

`LinkShareAccess`

Pointer to the share-access structure that describes how the open file object is currently being accessed.

Return Value

None

Remarks

IoUpdateLinkShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoUpdateLinkShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Before calling **IoUpdateLinkShareAccess**, the caller must successfully call IoCheckLinkShareAccess with *Update* set to False. Such a call to **IoCheckLinkShareAccess** determines whether the requested shared access is compatible with the way the file object is currently being accessed by other opens, but it does not update the **SHARE_ACCESS** structure. **IoUpdateLinkShareAccess** actually updates the **SHARE_ACCESS** structure associated with the file object.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Target Platform	Universal
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

IoCheckLinkShareAccess

IoRemoveLinkShareAccess

IoSetLinkShareAccess

IoUpdateShareAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **IoUpdateShareAccess** routine updates the share access for the given file object, usually when the file is being opened.

Syntax

```
void IoUpdateShareAccess(
  PFILE_OBJECT  FileObject,
  PSHARE_ACCESS ShareAccess
);
```

Parameters

`FileObject`

Pointer to a referenced file object representing the file or associated device object for which to update the share access.

`ShareAccess`

Pointer to the common **SHARE_ACCESS** structure associated with the *FileObject*. Drivers should treat this structure as opaque.

Return Value

None

Remarks

IoUpdateShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoUpdateShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Before calling **IoUpdateShareAccess**, the caller must successfully call **IoCheckShareAccess** with *Update* set to False. Such a call to **IoCheckShareAccess** determines whether the requested shared access is compatible with the way the file object is currently being accessed by other opens, but it does not update the **SHARE_ACCESS** structure. **IoUpdateShareAccess** actually updates the **SHARE_ACCESS** structure associated with the file object.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoCheckShareAccess

IoRemoveShareAccess

IoSetShareAccess

IoValidateDeviceIoControlAccess function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibIoValidateDeviceIoControlAccess function.

Syntax

```
NTSTATUS IoValidateDeviceIoControlAccess(
  PIRP  Irp,
  ULONG RequiredAccess
);
```

Parameters

`Irp`

For more information, see the WdmlibIoValidateDeviceIoControlAccess function.

`RequiredAccess`

For more information, see the WdmlibIoValidateDeviceIoControlAccess function.

Return Value

For more information, see the WdmlibIoValidateDeviceIoControlAccess function.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows. Drivers that must also work for Windows 2000 and Windows XP can instead link to Wdmsec.lib to use this routine. (The Wdmsec.lib library first shipped with the Windows XP Service Pack 1 [SP1] and Windows Server 2003 editions of the Driver Development Kit [DDK] and now ships with the Windows Driver Kit [WDK].)
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IRP

IoWithinStackLimits function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWithinStackLimits** routine determines whether a region of memory is within the stack limit of the current thread.

Syntax

```
LOGICAL IoWithinStackLimits(
  ULONG_PTR RegionStart,
  SIZE_T    RegionSize
);
```

Parameters

`RegionStart`

A pointer to the start of the region.

`RegionSize`

The size of the region.

Return Value

IoWithinStackLimits returns **TRUE** is the current thread's stack contains the region completely and **FALSE** otherwise.

Remarks

The **IoWithinStackLimits** routine considers all possible stack segments and the DPC stack, if necessary.

> **Note** For drivers that use KeExpandKernelStackAndCallout, **IoWithinStackLimits** considers only the current thread.

Requirements

Minimum supported client	Available on Windows Vista and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= APC_LEVEL"

See Also

IoGetInitialStack

IoGetRemainingStackSize

IoGetStackLimits

IoWMIAllocateInstanceIds function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIAllocateInstanceIds** routine allocates one or more instance IDs that are unique to the GUID.

Syntax

```
NTSTATUS IoWMIAllocateInstanceIds(
  LPCGUID Guid,
  ULONG   InstanceCount,
  ULONG   *FirstInstanceId
);
```

Parameters

`Guid`

Pointer to the GUID for which to generate instance identifiers.

`InstanceCount`

Specifies how many instance identifiers should be provided.

`FirstInstanceId`

Pointer to the first instance identifier that the driver should use.

Return Value

IoWMIAllocateInstanceIds returns a status code from the following list:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	Indicates that WMI successfully provided unique instance identifiers for the GUID specified.
STATUS_UNSUCCESSFUL	Indicates that the WMI services are not available.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	Indicates that insufficient resources were available to provide the caller with instance IDs.

Remarks

If greater than one instance was requested in *InstanceCount* and the routine completed successfully, *FirstInstanceId* points to the first instance that the caller should use. For each instance requested beyond one, the caller should increment the value returned in **FirstInstanceId*. For example, if the caller requested six instances and one was returned as the value of *FirstInstanceId*, the caller should use the values 1-6 as his unique instance identifiers.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, PowerIrpDDis, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

IoWmiSuggestInstanceName

IoWMIDeviceObjectToInstanceName function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIDeviceObjectToInstanceName** routine determines the instance name for the WMI class instance implemented by the driver that is specified by a device object.

Syntax

```
NTSTATUS IoWMIDeviceObjectToInstanceName(
  PVOID           DataBlockObject,
  PDEVICE_OBJECT  DeviceObject,
  PUNICODE_STRING InstanceName
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with IoWMIOpenBlock.

`DeviceObject`

Specifies a device object. The routine returns the instance name for the driver corresponding to the device object.

`InstanceName`

Pointer to the UNICODE_STRING structure that the routine uses to return the instance name. The caller frees the returned buffer within **UNICODE_STRING**.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine stores the instance name in the **UNICODE_STRING** specified by the *InstanceName* parameter.

STA **TUS** **_W** **MI_I** **NST** **AN** **CE_** **NO** **T_F** **OU** **ND**	The driver does not implement any instances of the WMI class specified by *DataBlockObject*.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DEVICE_OBJECT

IoWMIHandleToInstanceName

IoWMIOpenBlock

IoWMIDeviceObjectToProviderId function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIDeviceObjectToProviderId** routine translates the specified device object into the corresponding WMI Provider ID.

Syntax

```
ULONG IoWMIDeviceObjectToProviderId(
  PDEVICE_OBJECT DeviceObject
);
```

Parameters

`DeviceObject`

Pointer to a device object.

Return Value

IoWMIDeviceObjectToProviderId returns the WMI Provider ID associated with the specified device object.

Remarks

IoWMIDeviceObjectToProviderId should be used when filling in the **ProviderId** member of the WNODE_HEADER structure in those cases when the **WNODEHEADER** structure is being initialized as part of a WNODE_EVENT_ITEM or WNODE_EVENT_REFERENCE structure. (If the **WNODE_HEADER** is being used for other purposes, *ProviderId* is reserved.)

When running on a 32-bit operating system, the provider ID and the device object are identical. When running on a 64-bit operating system, **IoWMIDeviceObjectToProviderId** will convert the 64-bit device object to a 32-bit provider ID.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

WNODE_EVENT_ITEM

WNODE_EVENT_REFERENCE

WNODE_HEADER

IoWMIExecuteMethod function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIExecuteMethod** routine runs a WMI class method on the specified WMI data block instance.

Syntax

```
NTSTATUS IoWMIExecuteMethod(
  PVOID           DataBlockObject,
  PUNICODE_STRING InstanceName,
  ULONG           MethodId,
  ULONG           InBufferSize,
  PULONG          OutBufferSize,
  PUCHAR          InOutBuffer
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with IoWMIOpenBlock. The object must be opened with the WMIGUID_EXECUTE access right.

`InstanceName`

Specifies the name of the instance of the data block. This value corresponds to the value of the **InstanceName** property for the block.

`MethodId`

Specifies the method item ID for the method to be set. The value of this parameter corresponds to the value declared in the **WmiMethodId** qualifier for the method.

`InBufferSize`

Specifies the size, in bytes, of input data for the method. The actual input data is passed in the buffer pointed to by the *InOutBuffer* parameter.

`OutBufferSize`

Pointer to a ULONG that specifies the expected size, bytes, of the data output by the method. The actual output data is returned in the buffer pointed to by the *InOutBuffer* parameter. If the routine succeeds, it updates the memory location to specify the number of bytes actually stored in *InOutBuffer*. If the routine fails with STATUS_BUFFER_TOO_SMALL, it returns the number of bytes required to return the data.

`InOutBuffer`

Pointer to the buffer where the caller passes the WMI method's input data, and receives the WMI method's output data.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine returns the WMI data in the buffer pointed to by the *InOutBuffer* parameter. The routine also returns the size, in bytes, of the returned data in the memory location pointed to by the *OutBufferSize* parameter.
STATUS_WMI_GUID_NOT_FOUND	No drivers implement the WMI class.
STATUS_WMI_INSTANCE_NOT_FOUND	No driver implements an instance of the WMI class with **InstanceName** property equal to the value specified in the *InstanceName* parameter.
STATUS_WMI_ITEMID_NOT_FOUND	The WMI class does not contain a method with method ID equal to the value of *MethodId*.
STATUS_BUFFER_TOO_SMALL	The buffer passed by the caller in the *OutBuffer* parameter is too small. The routine returns the required buffer size in the memory location pointed to by the *InOutBufferSize* parameter.

STA **TUS** **_W** **MI_** **GUI** **D_D** **ISC** **ON** **NEC** **TED**	The WMI GUID is no longer available or was never available.

Remarks

IoWMIExecuteMethod determines which drivers might support the specified WMI class and instance name. It issues an IRP_MN_EXECUTE_METHOD request to each such driver. The driver that exports the data block instance with matching **InstanceName** property then runs the specified WMI method.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_EXECUTE_METHOD

IoWMIOpenBlock

IoWMIHandleToInstanceName function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIHandleToInstanceName** routine determines the instance name for the WMI class instance implemented by the driver that is specified by a file handle.

Syntax

```
NTSTATUS IoWMIHandleToInstanceName(
  PVOID           DataBlockObject,
  HANDLE          FileHandle,
  PUNICODE_STRING InstanceName
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with the IoWMIOpenBlock routine.

`FileHandle`

Specifies a file handle. The routine returns the instance name corresponding to the driver that is represented by the file handle.

`InstanceName`

Pointer to the UNICODE_STRING structure that the routine uses to return the instance name. The caller frees the returned buffer within **UNICODE_STRING**.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine fills the **UNICODE_STRING** structure pointed to by the *InstanceName* parameter with the instance name.

STA **TUS** **_W** **MI_I** **NST** **AN** **CE_** **NO** **T_F** **OU** **ND**	The driver does not implement any instances of the WMI class specified by *DataBlockObject*.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DEVICE_OBJECT

IoWMIDeviceObjectToInstanceName

IoWMIOpenBlock

IoWMIOpenBlock function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIOpenBlock** routine opens the WMI data block object for the specified WMI class.

Syntax

```
NTSTATUS IoWMIOpenBlock(
  LPCGUID Guid,
  ULONG   DesiredAccess,
  PVOID   *DataBlockObject
);
```

Parameters

`Guid`

`DesiredAccess`

Specifies the desired access rights to the data block object. The caller must have particular access rights to perform certain operations.

The following is a description of each access right bit and the operations it allows:

WMIGUID_EXECUTE

The data block object can be used to run WMI class methods. This flag must be set to use IoWMIExecuteMethod on the data block object.

WMIGUID_NOTIFICATION

The data block object can be used to register event notification callbacks. This flag must be set to use IoWMISetNotificationCallback, and can only be used for WMI event blocks. Callers that specify this flag must also specify the SYNCHRONIZE flag.

WMIGUID_QUERY

The data block object can be used to query WMI class properties. This flag must be set to use any of the **IoWMIQuery*Xxx*** routines on the data block object.

WMIGUID_SET

The data block object can be used to set WMI class properties. This flag must be set to use any of the **IoWMISet*Xxx*** routines on the data block object.

`DataBlockObject`

Pointer to a memory location where the routine returns a pointer to the data block object.

Return Value

Returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

The caller uses **IoWMIOpenBlock** to create a data block object for the specified WMI class GUID. Subsequently, the caller can use the data block object to read or write WMI class properties, run WMI class methods, and register

callbacks for WMI events.

Use the **IoWMIQuery*Xxx*** and **IoWMISet*Xxx*** routines to read and write WMI class properties. Use IoWMIExecuteMethod to run WMI class methods, and use IoWMISetNotificationCallback to register a WMI event notification callback.

Use ObDereferenceObject to close the data block object once it is no longer needed.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Ntifs.h. The WMIGUID_XXX values are declared in Wmistr.h. To use them, include Wmistr.h.)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IoWMIExecuteMethod

IoWMIQueryAllData

IoWMIQueryAllDataMultiple

IoWMIQuerySingleInstance

IoWMIQuerySingleInstanceMultiple

IoWMISetNotificationCallback

IoWMISetSingleInstance

IoWMISetSingleItem

ZwClose

IoWMIQueryAllData function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIQueryAllData** routine returns all WMI data blocks that implement a given WMI class.

Syntax

```
NTSTATUS IoWMIQueryAllData(
  PVOID  DataBlockObject,
  PULONG InOutBufferSize,
  PVOID  OutBuffer
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with the IoWMIOpenBlock routine. The object must be opened with the WMIGUID_QUERY access right.

`InOutBufferSize`

Pointer to a memory location that specifies the size of the buffer passed in the *OutBuffer* parameter. If the routine succeeds, it updates the memory location to specify the number of bytes actually stored in *OutBuffer*. If the routine fails with status code of STATUS_BUFFER_TOO_SMALL, it returns the number of bytes required to return the data.

`OutBuffer`

Pointer to the buffer where the routine returns the WMI data. The routine returns a sequence of variable-sized WNODE_ALL_DATA structures, one for each set of returned data blocks. The **WnodeHeader.Linkage** member of each **WNODE_ALL_DATA** structure contains the offset from the beginning of the current **WNODE_ALL_DATA** to the beginning of the next **WNODE_ALL_DATA**. The final block in the chain has **WnodeHeader.Linkage** set to zero. *OutBuffer* must point to a buffer allocated from nonpaged pool.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine returns the WMI data in the buffer pointed to by the *OutBuffer* parameter. The routine also returns the size, in bytes, of the returned data in the memory location pointed to by the *InOutBufferSize* parameter.

STA **TUS** **_W** **MI_** **GUI** **D_N** **OT_** **FOU** **ND**	No drivers implement the specified WMI class.
STA **TUS** **_BU** **FFE** **R_T** **OO_** **SM** **ALL**	The buffer passed by the caller in the *OutBuffer* parameter is too small. The routine returns the required buffer size in the memory location pointed to by the *InOutBufferSize* parameter.

Remarks

IoWMIQueryAllData determines which drivers support the specified WMI class, and issues an **IRP_MN_QUERY_ALL_DATA** request to every such driver.

To query for multiple WMI classes, use IoWMIQueryAllDataMultiple.

Requirements

Minimum supported client	Available in Microsoft Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_QUERY_ALL_DATA

IoWMIOpenBlock

IoWMIQueryAllDataMultiple

IoWMIQueryAllDataMultiple function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIQueryAllDataMultiple** routine returns all WMI data blocks that implement one of a set of WMI classes.

Syntax

```
NTSTATUS IoWMIQueryAllDataMultiple(
  PVOID  *DataBlockObjectList,
  ULONG  ObjectCount,
  PULONG InOutBufferSize,
  PVOID  OutBuffer
);
```

Parameters

`DataBlockObjectList`

Pointer to an array of pointers to WMI data block objects. The caller opens a data block object for each WMI class with the IoWMIOpenBlock routine. Each object must be opened with the WMIGUID_QUERY access right.

`ObjectCount`

Specifies the number of entries in the array passed in the *DataBlockObjectList* parameter.

`InOutBufferSize`

Pointer to a memory location that specifies the size of the buffer passed in the *OutBuffer* parameter. If the routine succeeds, it updates the memory location to specify the number of bytes actually stored in *OutBuffer*. If the routine fails with status code of STATUS_BUFFER_TOO_SMALL, it returns the number of bytes required to return the data.

`OutBuffer`

Pointer to the buffer where the routine returns the WMI data. The routine returns a sequence of variable-sized WNODE_ALL_DATA structures, one for each set of returned data blocks. The **WnodeHeader.Linkage** member of each **WNODE_ALL_DATA** structure contains the offset from the beginning of the current **WNODE_ALL_DATA** to the beginning of the next **WNODE_ALL_DATA**. The final block in the chain has **WnodeHeader.Linkage** set to zero. *OutBuffer* must point to a buffer allocated from nonpaged pool.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine returns the WMI data in the buffer pointed to by the *OutBuffer* parameter. The routine also returns the size, in bytes, of the returned data in the memory location pointed to by the *InOutBufferSize* parameter.

STA **TUS** **_BU** **FFE** **R_T** **OO_** **SM** **ALL**	The buffer passed by the caller in the *OutBuffer* parameter is too small. The routine returns the required buffer size in the memory location pointed to by the *InOutBufferSize* parameter.

Remarks

IoWMIQueryAllDataMultiple determines which drivers support the specified WMI classes, and issues an IRP_MN_QUERY_ALL_DATA request to every such driver.

If no drivers implement any of the specified WMI classes, the routine returns STATUS_SUCCESS It also returns a value of zero in the memory location pointed to by the *InOutBufferSize* parameter.

To query a single WMI class, use IoWMIQueryAllData.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_QUERY_ALL_DATA

IoWMIOpenBlock

IoWMIQueryAllData

IoWMIQuerySingleInstance function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIQuerySingleInstance** routine returns the specified instance of a WMI data block.

Syntax

```
NTSTATUS IoWMIQuerySingleInstance(
  PVOID           DataBlockObject,
  PUNICODE_STRING InstanceName,
  PULONG          InOutBufferSize,
  PVOID           OutBuffer
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with the IoWMIOpenBlock routine. The object must be opened with the WMIGUID_QUERY access right.

`InstanceName`

Specifies the name of the instance of the data block. This value corresponds to the value of the **InstanceName** property for the block.

`InOutBufferSize`

Pointer to a memory location that specifies the size of the buffer passed in the *OutBuffer* parameter. If the routine succeeds, it updates the memory location to specify the number of bytes actually stored in *OutBuffer*. If the routine fails with status code of STATUS_BUFFER_TOO_SMALL, it returns the number of bytes required to return the data.

`OutBuffer`

Pointer to the buffer where the routine returns the WMI data. The routine returns a variable-sized WNODE_SINGLE_INSTANCE structure. *OutBuffer* must point to a buffer allocated from nonpaged pool.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The routine returns the WMI data in the buffer pointed to by the *OutBuffer* parameter. The routine also returns the size, in bytes, of the returned data in the memory location pointed to by the *InOutBufferSize* parameter.

STA TUS _W MI_ GUI D_N OT_ FOU ND	No drivers implement the specified WMI class.
STA TUS _W MI_I NST AN CE_ NO T_F OU ND	No driver implements an instance of the WMI class with **InstanceName** property equal to the value specified in the *InstanceName* parameter.
STA TUS _BU FFE R_T OO_ SM ALL	The buffer passed by the caller in the *OutBuffer* parameter is too small. The routine returns the required buffer size in the memory location pointed to by the *InOutBufferSize* parameter.

Remarks

IoWMIQuerySingleInstance determines which drivers might support the specified WMI class with the specified instance name, and issues an IRP_MN_QUERY_SINGLE_INSTANCE request to each such driver. The driver that exports the data block instance with matching **InstanceName** property returns the appropriate data.

To query for multiple WMI classes and instance names, use the IoWMIQuerySingleInstanceMultiple routine. Drivers can use the IoWMISetSingleInstance routine to update the class instance.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_QUERY_SINGLE_INSTANCE

IoWMIOpenBlock

IoWMIQuerySingleInstanceMultiple

IoWMISetSingleInstance

IoWMIQuerySingleInstanceMultiple function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIQuerySingleInstanceMultiple** routine returns all WMI data block instances that implement the specified WMI classes with the specified instance names.

Syntax

```
NTSTATUS IoWMIQuerySingleInstanceMultiple(
  PVOID           *DataBlockObjectList,
  PUNICODE_STRING InstanceNames,
  ULONG           ObjectCount,
  PULONG          InOutBufferSize,
  PVOID           OutBuffer
);
```

Parameters

`DataBlockObjectList`

Pointer to an array of pointers of WMI data block objects. The caller opens a data block object for each WMI class with the IoWMIOpenBlock routine. Each object must be opened with the WMIGUID_QUERY access right.

`InstanceNames`

Pointer to an array of UNICODE_STRING structures containing instance names. The *n*th instance name in the array corresponds to the value of the **InstanceName** property for the *n*th WMI class specified in the array pointed to by the *DataBlockObjectList* parameter.

`ObjectCount`

Specifies the number of entries in the arrays passed in the *DataBlockObjectList* and *InstanceNames* parameters.

`InOutBufferSize`

Pointer to a memory location that specifies the size of the buffer passed in the *OutBuffer* parameter. If the routine succeeds, it updates the memory location to specify the number of bytes actually stored in *OutBuffer*. If the routine fails with STATUS_BUFFER_TOO_SMALL, it returns the number of bytes required to return the data.

`OutBuffer`

Pointer to the buffer where the routine returns the WMI data. The routine returns a sequence of variable-sized WNODE_SINGLE_INSTANCE structures, one for each data block instance. The **WnodeHeader.Linkage** member of each **WNODE_SINGLE_INSTANCE** structure contains the offset from the beginning of the current **WNODE_SINGLE_INSTANCE** to the beginning of the next **WNODE_SINGLE_INSTANCE**. The final block in the chain has **WnodeHeader.Linkage** set to zero. Each distinct data block instance corresponds to a single matching WMI class and instance name. *OutBuffer* must point to a buffer allocated from nonpaged pool.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION

STA TUS _SU CCE SS	The operation succeeded. The routine returns the WMI data in the buffer pointed to by the *OutBuffer* parameter. The routine also returns the size, in bytes, of the returned data in the memory location pointed to by the *InOutBufferSize* parameter.
STA TUS _BU FFE R_T OO_ SM ALL	The buffer passed by the caller in the *OutBuffer* parameter is too small. The routine returns the required buffer size in the memory location pointed to by the *InOutBufferSize* parameter.

Remarks

IoWMIQuerySingleInstanceMultiple determines which drivers might support the specified WMI classes and instance names, and issues an IRP_MN_QUERY_SINGLE_INSTANCE request to each such driver. Each driver that exports the data block instance with matching **InstanceName** property returns the appropriate data.

If no drivers implement any of the specified WMI classes and instance names, the routine returns STATUS_SUCCESS. It also returns a value of zero in the memory location pointed to by the *InOutBufferSize* parameter.

To query for a single WMI class and instance name, use the IoWMIQuerySingleInstance routine. Drivers can use the IoWMISetSingleInstance routine to update a class instance.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_QUERY_SINGLE_INSTANCE

IoWMIOpenBlock

IoWMIQuerySingleInstance

IoWMISetSingleInstance

UNICODE_STRING

IoWMIRegistrationControl function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIRegistrationControl** routine registers or unregisters the caller as a WMI data provider for a specified device object.

Syntax

```
NTSTATUS IoWMIRegistrationControl(
  PDEVICE_OBJECT DeviceObject,
  ULONG          Action
);
```

Parameters

`DeviceObject`

A pointer to a device object. This object is a DEVICE_OBJECT system structure.

`Action`

The action that WMI should take. The requested action is indicated by one of the following values.

ACTION VALUE	MEANING
WMIREG_ACTION_REGISTER	Specifies that WMI should register the caller as a WMI provider for *DeviceObject*. This will result in WMI sending an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request to the driver.
WMIREG_ACTION_DEREGISTER	Specifies that WMI should remove the caller from its list of WMI providers for *DeviceObject*.
WMIREG_ACTION_REREGISTER	Specifies that WMI should unregister the driver and then register (reregister) the driver. Reregistering the driver results in WMI sending an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request to the driver.
WMIREG_ACTION_UPDATE_GUIDS	Specifies that WMI should re-query the driver for a new list of GUID identifiers that it provides data for. This will result in WMI sending an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request to the driver.

Return Value

IoWMIRegistrationControl returns a status code from the following list:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	Indicates that WMI completed the action requested without error.
STATUS_INVALID_PARAMETER	Indicates that the action, specified in *Action*, was invalid.
STATUS_*XXX*	Indicates that the request failed for the reason specified by the NTSTATUS value. See Ntstatus.h for detailed information for the actual status return code.

Remarks

After a driver calls **IoWMIRegistrationControl**, WMI sends the driver an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request so the driver can provide information to WMI. For more information, see Registering as a WMI Data Provider.

If the caller specifies WMIREG_ACTION_DEREGISTER for *Action*, **IoWMIRegistrationControl** causes the calling thread to block until all **IRP_MJ_SYSTEM_CONTROL** requests that were previously sent to the specified device object have completed. In such a case, if a driver calls **IoWMIRegistrationControl** within a dispatch routine for an **IRP_MJ_SYSTEM_CONTROL** request, the calling thread will deadlock.

If a device is removed suddenly (for example, in a surprise removal), causing the PnP manager to send an IRP_MN_SURPRISE_REMOVAL IRP, the driver must call **IoWMIRegistrationControl** and specify WMIREG_ACTION_DEREGISTER in *Action* in the call. Note that if the driver calls **IoWMIRegistrationControl** with *Action* set to WMIREG_ACTION_DEREGISTER in response to an **IRP_MN_SURPRISE_REMOVAL** IRP, the driver must not make the same call to **IoWMIRegistrationControl** in response to an IRP_MN_REMOVE_DEVICE IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlIoPassive5, LowerDriverReturn, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IRP_MN_REGINFO

IRP_MN_REGINFO_EX

IRP_MN_REMOVE_DEVICE

IRP_MN_SURPRISE_REMOVAL

IoWMISetNotificationCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMISetNotificationCallback** routine registers a notification callback for a WMI event.

Syntax

```
NTSTATUS IoWMISetNotificationCallback(
  PVOID                    Object,
  WMI_NOTIFICATION_CALLBACK Callback,
  PVOID                    Context
);
```

Parameters

`Object`

Pointer to a WMI data block object. The caller opens the data block object for the WMI event with the IoWMIOpenBlock routine. The object must be opened with the WMIGUID_NOTIFICATION access right.

`Callback`

Pointer to a function of the form:

```
XxxWmiNotificationCallback(PVOID Wnode, PVOID Context);
```

WMI calls this function to notify the caller that the specified event has occurred. The *Wnode* parameter of the callback routine points to the WNODE_EVENT_ITEM structure returned by the driver triggering the event. The *Context* parameter of the callback routine points to the value specified in the *Context* parameter of the **IoWMISetNotificationCallback** routine.

`Context`

Specifies the value that WMI passes to the callback routine when the event occurs.

Return Value

This routine returns STATUS_SUCCESS on success, and the appropriate NTSTATUS error code on failure.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IoWMIOpenBlock

WNODE_EVENT_ITEM

IoWMISetSingleInstance function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMISetSingleInstance** routine sets the values for properties within the data block instance that matches the specified WMI class and instance name.

Syntax

```
NTSTATUS IoWMISetSingleInstance(
  PVOID           DataBlockObject,
  PUNICODE_STRING InstanceName,
  ULONG           Version,
  ULONG           ValueBufferSize,
  PVOID           ValueBuffer
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with the IoWMIOpenBlock routine. The object must be opened with the WMIGUID_SET access right.

`InstanceName`

Specifies the name of the instance of the data block. This value corresponds to the value of the **InstanceName** property for the block.

`Version`

Reserved for future use. Callers must set this parameter to zero.

`ValueBufferSize`

Specifies the size, in bytes, of the buffer passed in the *ValueBuffer* parameter.

`ValueBuffer`

Pointer to the buffer that contains the new values for the properties within the data block.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded. The values of the WMI data block instance properties are updated to the contents of the buffer pointed to by the *ValueBuffer* parameter.

STATUS_WMI_GUID_NOT_FOUND	No drivers implement the WMI class.
STATUS_WMI_INSTANCE_NOT_FOUND	No driver implements an instance of the WMI class with **InstanceName** property equal to the value specified in the *InstanceName* parameter.
STATUS_WMI_READ_ONLY	All properties of the WMI class are read-only.
STATUS_WMI_SET_FAILURE	The driver that implements the WMI data block instance is unable to update the instance.

Remarks

IoWMISetSingleInstance determines which drivers might support the specified WMI class and instance name, and issues an IRP_MN_CHANGE_SINGLE_INSTANCE request to each such driver. The driver that exports the data block instance with matching **InstanceName** property updates its data block instance. Note that a data block might have both read-only and read/write properties. In this case, only the read/write properties will be updated and STATUS_SUCCESS is returned.

Drivers can also use the IoWMISetSingleItem routine to update a single property within the class instance.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_CHANGE_SINGLE_INSTANCE

IoWMIOpenBlock

IoWMIQuerySingleInstance

IoWMISetSingleItem

IoWMISetSingleItem function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMISetSingleItem** routine sets a single property in the data block instance that matches the specified WMI class and instance name.

Syntax

```
NTSTATUS IoWMISetSingleItem(
  PVOID           DataBlockObject,
  PUNICODE_STRING InstanceName,
  ULONG           DataItemId,
  ULONG           Version,
  ULONG           ValueBufferSize,
  PVOID           ValueBuffer
);
```

Parameters

`DataBlockObject`

Pointer to a WMI data block object. The caller opens the data block object for the WMI class with the IoWMIOpenBlock routine. The object must be opened with the WMIGUID_SET access right.

`InstanceName`

Specifies the name of the instance of the data block. This value corresponds to the value of the **InstanceName** property for the block.

`DataItemId`

Specifies the data item ID for the property to be set. The value of this parameter corresponds to the value declared in the **WmiDataId** qualifier for the property.

`Version`

Reserved for future use. Callers must set this parameter to zero.

`ValueBufferSize`

Specifies the size, in bytes, of the buffer passed in the *ValueBuffer* parameter.

`ValueBuffer`

Pointer to the buffer that contains the new value for the property specified by the *DataItemId* parameter.

Return Value

The routine returns an NTSTATUS code. Possible return values include:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The operation succeeded. The value of the property within WMI data block instance is updated to the contents of the buffer pointed to by the *ValueBuffer* parameter.
STATUS_WMI_GUID_NOT_FOUND	No drivers implement the WMI class.
STATUS_WMI_INSTANCE_NOT_FOUND	No driver implements an instance of the WMI data block with **InstanceName** property equal to the value specified in the *InstanceName* parameter.
STATUS_WMI_ITEMID_NOT_FOUND	The WMI class does not contain a property with data item ID equal to the value of *DataItemId*.
STATUS_WMI_READ_ONLY	The data item ID in the data block is read-only.

STATUS_WMI_SET_FAILURE	The driver that implements the WMI data block instance is unable to update the property specified by *DataItemId*.

Remarks

IoWMISetSingleItem determines which drivers might support the specified WMI class and instance name. It issues an IRP_MN_CHANGE_SINGLE_ITEM request to each such driver. The driver that exports the data block instance with matching **InstanceName** property updates the specified property in the data block instance.

Drivers can also use IoWMISetSingleInstance to update every property of the WMI class instance simultaneously.

Requirements

Minimum supported client	Available in Windows XP and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP_MN_CHANGE_SINGLE_ITEM

IoWMIOpenBlock

IoWMIQuerySingleInstance

IoWMISetSingleInstance

IoWMISuggestInstanceName function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMISuggestInstanceName** routine is used to request that WMI suggest a base name which a driver can use to build WMI instance names for the device.

Syntax

```
NTSTATUS IoWMISuggestInstanceName(
  PDEVICE_OBJECT  PhysicalDeviceObject,
  PUNICODE_STRING SymbolicLinkName,
  BOOLEAN         CombineNames,
  PUNICODE_STRING SuggestedInstanceName
);
```

Parameters

`PhysicalDeviceObject`

If supplied, points to the driver's physical device object.

`SymbolicLinkName`

If supplied, points to the symbolic link name returned from IoRegisterDeviceInterface.

`CombineNames`

If **TRUE** then the suggested names returned will combine the *PhysicalDeviceObject* and *SymbolicLinkName* information.

`SuggestedInstanceName`

A pointer to a buffer which upon successful completion will contain a UNICODE_STRING which contains the suggested instance name. The caller is responsible for freeing this buffer when it is no longer needed.

Return Value

IoWMISuggestInstanceName returns a status code from the following list:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	Indicates that WMI was able to successfully complete this function.

STATUS_UNSUCCESSFUL	Indicates that the WMI services are not available.
STATUS_INSUFFICIENT_RESOURCES	Indicates that insufficient resources were available to provide the caller with a buffer containing the Unicode string.
STATUS_NO_MEMORY	Indicates that insufficient resources were available to provide the caller with a buffer containing the Unicode string.

Remarks

If the *CombineNames* parameter is **TRUE** then both *PhysicalDeviceObject* and *SymbolicLinkName* must be specified. Otherwise, only one of them should be specified.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IoWMIAllocateInstanceIds

IoWMIWriteEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWMIWriteEvent** routine delivers a given event to the user-mode WMI components for notification.

Syntax

```
NTSTATUS IoWMIWriteEvent(
  PVOID WnodeEventItem
);
```

Parameters

`WnodeEventItem`

Pointer to a WNODE_EVENT_ITEM structure to be delivered to the user-mode WMI components that requested notification of the event.

Return Value

IoWMIWriteEvent returns a status code from the following list:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	Indicates that WMI has successfully queued the event for delivery to the user-mode WMI components.
STATUS_UNSUCCESSFUL	Indicates that WMI services are unavailable.
STATUS_BUFFER_OVERFLOW	Indicates that the event item specified exceeds the maximum allowed size.

STA TUS _IN SUF FICI ENT _RE SO URC ES	Indicates that insufficient resources were available for WMI to queue the event for delivery.

Remarks

The WNODE_EVENT_ITEM structure that is allocated by the caller and passed in *WnodeEventItem* must be allocated from nonpaged pool. If **IoWMIWriteEvent** returns STATUS_SUCCESS, the memory for the event item will automatically be freed by the system. If **IoWMIWriteEvent** returns anything other than STATUS_SUCCESS, it is the caller's responsibility to free the buffer.

Drivers should only call **IoWMIWriteEvent** for events that have been enabled for WMI. This ensures that there is an event consumer waiting for indication on that event.

Callers of this routine must be running at IRQL <= APC_LEVEL, with one exception. When the **Flags** member of the WNODE_HEADER structure contains WNODE_FLAG_TRACED_GUID, **IoWMIWriteEvent** can be called at any IRQL. (The **WNODE_HEADER** structure is a member of the WNODE_EVENT_ITEM structure that the *WnodeEventItem* parameter points to.)

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (see Remarks section)"

See Also

IoWmiDeviceObjectToProviderId

WNODE_EVENT_ITEM

IoWriteErrorLogEntry function

1/25/2019 • 2 minutes to read • Edit Online

The **IoWriteErrorLogEntry** routine queues a given error log packet to the system error logging thread.

Syntax

```
void IoWriteErrorLogEntry(
  PVOID ElEntry
);
```

Parameters

`ElEntry`

Pointer to the error log packet the driver has allocated with IoAllocateErrorLogEntry and filled in by the caller.

Return Value

None

Remarks

IoWriteErrorLogEntry frees the error log entry. Drivers must not call **IoFreeErrorLogEntry** on a log entry that they have already passed to **IoWriteErrorLogEntry**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

IO_ERROR_LOG_PACKET

IoAllocateErrorLogEntry

IoFreeErrorLogEntry

IsListEmpty function

1/25/2019 • 2 minutes to read • Edit Online

The **IsListEmpty** routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.

Syntax

```
BOOLEAN IsListEmpty(
  const LIST_ENTRY *ListHead
);
```

Parameters

`ListHead`

Pointer to a LIST_ENTRY structure that represents the head of the list.

Return Value

IsListEmpty returns **TRUE** if there are currently no entries in the list and **FALSE** otherwise.

Remarks

IsListEmpty returns **TRUE** if *ListHead*->**Flink** refers back to *ListHead*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **IsListEmpty** can be running at any IRQL. If **IsListEmpty** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

InitializeListHead

RemoveEntryList

RemoveHeadList

RemoveTailList

KBUGCHECK_CALLBACK_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *BugCheckCallback* routine is executed whenever the system issues a bug check.

Syntax

```
KBUGCHECK_CALLBACK_ROUTINE KbugcheckCallbackRoutine;

void KbugcheckCallbackRoutine(
  IN PVOID Buffer,
  IN ULONG Length
)
{...}
```

Parameters

`Buffer`

A pointer to the buffer that was specified when the callback was registered.

`Length`

Specifies the length, in bytes, of the buffer that is pointed to by the *Buffer* parameter.

Return Value

None

Remarks

Drivers can supply a *BugCheckCallback* that resets the device to a known state if the system issues a bug check.

Use KeRegisterBugCheckCallback to register a *BugCheckCallback* routine. A driver can subsequently remove the callback by using the KeDeregisterBugCheckCallback routine. If the driver can be unloaded, it must remove any registered callbacks in its Unload routine.

A *BugCheckCallback* routine is strongly restricted in the actions it can take. For more information, see Writing a Bug Check Callback Routine. The routine can safely use the **READ_PORT_*XXX***, **READ_REGISTER_*XXX***, **WRITE_PORT_*XXX***, and **WRITE_REGISTER_*XXX*** routines to interact with the device.

Drivers that require more sophisticated interaction with the system as it issues a bug check can instead implement BugCheckDumpIoCallback or BugCheckSecondaryDumpDataCallback routines.

Note that beginning with the Windows XP SP1 and Windows Server 2003 operating systems, *BugCheckCallback* routines execute after the system crash dump file has already been written. (On earlier versions of Windows, the routines execute before the crash dump file is written.) Thus, any data that is stored in the buffer specified by the *Buffer* parameter will not appear in the crash dump file. Drivers that are required to write data to the crash dump file instead implement a BugCheckSecondaryDumpDataCallback routine. (On earlier versions of Windows, the data written to *Buffer* does appear in the crash dump file.)

Examples

To define a *BugCheckCallback* callback routine, you must first provide a function declaration that identifies the type

of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *BugCheckCallback* callback routine that is named `MyBugCheckCallback`, use the KBUGCHECK_CALLBACK_ROUTINE type as shown in this code example:

```
KBUGCHECK_CALLBACK_ROUTINE MyBugCheckCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyBugCheckCallback(
    PVOID  Buffer,
    ULONG  Length
    )
  {
      // Function body
  }
```

The KBUGCHECK_CALLBACK_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KBUGCHECK_CALLBACK_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at HIGH_LEVEL.

See Also

BugCheckDumpIoCallback

BugCheckSecondaryDumpDataCallback

KeDeregisterBugCheckCallback

KeRegisterBugCheckCallback

KBUGCHECK_REASON_CALLBACK_ROUTINE callback function

1/25/2019 • 9 minutes to read • Edit Online

Driver-implemented callback functions that the system executes when it issues a bug check.

The *BugCheckAddPagesCallback* routine adds one or more pages of data to the crash dump file when the operating system issues a bug check.

The *BugCheckDumpIoCallback* routine is executed each time the system writes data to a crash dump file.

The *BugCheckSecondaryDumpDataCallback* routine provides data to the system to append to the crash dump file when the system issues a bug check.

Syntax

```
KBUGCHECK_REASON_CALLBACK_ROUTINE KbugcheckReasonCallbackRoutine;

void KbugcheckReasonCallbackRoutine(
  KBUGCHECK_CALLBACK_REASON Reason,
  _KBUGCHECK_REASON_CALLBACK_RECORD *Record,
  PVOID ReasonSpecificData,
  ULONG ReasonSpecificDataLength
)
{...}
```

Parameters

`Reason`

Specifies the reason for the call to the callback routine. This value can be one of the following as defined in KBUGCHECK_CALLBACK_REASON

- **KbCallbackAddPages**: The purpose of the callback function is to enable the driver to add pages of data to the crash dump file.
- **KbCallbackDumpIo**: This callback function is executed each time the system writes data to a crash dump file.
- **KbCallbackSecondaryDumpData**: This callback function provides data to the system to append to the crash dump file when the system issues a bug check.

`Record`

A pointer to the KBUGCHECK_REASON_CALLBACK_RECORD structure that the driver supplied when it registered this callback. For more information, see the description of the *CallbackRecord* parameter in KeRegisterBugCheckCallback.

`ReasonSpecificData`

A pointer to a KBUGCHECK_ADD_PAGES structure. This pointer is cast to type PVOID. Certain members of this structure are filled in by the operating system before it calls the callback routine, and other members must be filled in by the callback routine. For more information, see the following Remarks section.

`ReasonSpecificDataLength`

Specifies the size, in bytes, of the buffer that the *ReasonSpecificData* parameter points to.

- **KbCallbackAddPages**: **sizeof**(**KBUGCHECK_ADD_PAGES**).
- **KbCallbackDumpIo**: **sizeof**(**KBUGCHECK_DUMP_IO**).
- **KbCallbackSecondaryDumpData**: **sizeof**(**KBUGCHECK_SECONDARY_DUMP_DATA**).

Return Value

None.

Remarks

About implementing *BugCheckAddPagesCallback*: A kernel-mode driver can implement a *BugCheckAddPagesCallback* callback routine to add one or more pages of data to a crash dump file when a bug check occurs. To register this routine with the operating system, the driver calls the KeRegisterBugCheckReasonCallback routine. Before the driver unloads, it must call the KeDeregisterBugCheckReasonCallback routine to remove the registration.

Starting with Windows 8, a registered *BugCheckAddPagesCallback* routine is called during a kernel memory dump or a complete memory dump. In earlier versions of Windows, a registered *BugCheckAddPagesCallback* routine is called during a kernel memory dump, but not during a complete memory dump. By default, a kernel memory dump includes only the physical pages that are being used by the Windows kernel at the time that the bug check occurs, whereas a complete memory dump includes all of the physical memory that is used by Windows. A complete memory dump does not, by default, include physical memory that is used by the platform firmware.

Your *BugCheckAddPagesCallback* routine can supply driver-specific data to add to the dump file. For example, for a kernel memory dump, this additional data can include physical pages that are not mapped to the system address range in virtual memory but that contain information that can help you to debug your driver. The *BugCheckAddPagesCallback* routine might add to the dump file any driver-owned physical pages that are unmapped or that are mapped to user-mode addresses in virtual memory.

When a bug check occurs, the operating system calls all the registered *BugCheckAddPagesCallback* routines to poll drivers for data to add to the crash dump file. Each call adds one or more pages of contiguous data to the crash dump file. A *BugCheckAddPagesCallback* routine can supply either a virtual address or a physical address for the starting page. If more than one page is supplied during a call, the pages are contiguous in either virtual or physical memory, depending on whether the starting address is virtual or physical. To supply noncontiguous pages, the *BugCheckAddPagesCallback* routine can set a flag in the **KBUGCHECK_ADD_PAGES** structure to indicate that it has additional data and has to be called again. For more information, see KBUGCHECK_ADD_PAGES.

Unlike a BugCheckSecondaryDumpDataCallback routine, which appends data to the secondary crash dump region, a *BugCheckAddPagesCallback* routine adds pages of data to the primary crash dump region. During debugging, primary crash dump data is easier to access than secondary crash dump data.

Before the operating system calls a *BugCheckAddPagesCallback* routine, it fills in the **BugCheckCode** member of the **KBUGCHECK_ADD_PAGES** structure that *ReasonSpecificData* points to. During the call, the *BugCheckAddPagesCallback* routine must set the values of the **Flags**, **Address**, and **Count** members of this structure. If the *BugCheckAddPagesCallback* routine is called more than one time, the operating system preserves the value that the callback routine wrote to the **Context** member in the previous call. Before the first call, the operating system initializes **Context** to **NULL**.

A *BugCheckAddPagesCallback* routine is very restricted in the actions it can take. For more information, see Writing a Bug Check Callback Routine.

About implementing *BugCheckDumpIoCallback*: The driver's *BugCheckDumpIoCallback* routine is called each time data is written to the crash dump file. The system passes, in the *ReasonSpecificData* parameter, a description of the data being written. The **Buffer** member points to the current data, and the **BufferLength** member specifies

its length. The **Type** member indicates the type of data currently being written, such as dump file header information, memory state, or data provided by a driver. For a description of the possible types of information, see KBUGCHECK_DUMP_IO_TYPE.

The system can write the crash dump file either sequentially, or out of order. If the system is writing the crash dump file sequentially, then the **Offset** member of *ReasonSpecificData* is -1; otherwise, **Offset** is set to the current offset, in bytes, in the crash dump file.

When the system writes the file sequentially, it calls each *BugCheckDumpIoCallback* routine one or more times when writing the header information (**Type** = **KbDumpIoHeader**), one or more times when writing the main body of the crash dump file (**Type** = **KbDumpIoBody**), and one or more times when writing the secondary dump data (**Type** = **KbDumpIoSecondaryDumpData**). Once the system has completed writing the crash dump file, it calls the callback with **Buffer** = **NULL**, **BufferLength** = 0, and **Type** = **KbDumpIoComplete**.

The main purpose of a *BugCheckDumpIoCallback* routine is to allow system crash dump data to be written to devices other than the disk. For example, a device that monitors system state can use the callback to report that the system has issued a bug check, and to provide a crash dump for analysis.

Use KeRegisterBugCheckReasonCallback to register a *BugCheckDumpIoCallback* routine. A driver can subsequently remove the callback by using the KeDeregisterBugCheckReasonCallback routine. If the driver can be unloaded, it must remove any registered callbacks in its Unload routine.

A *BugCheckDumpIoCallback* routine is strongly restricted in the actions it can take. For more information, see Writing a Bug Check Callback Routine.

About implementing *BugCheckSecondaryDumpDataCallback*: The system uses *BugCheckSecondaryDumpDataCallback* routines to poll drivers for crash dump data.

The system sets the **InBuffer**, **InBufferLength**, **OutBuffer**, and **MaximumAllowed** members of the KBUGCHECK_SECONDARY_DUMP_DATA structure that *ReasonSpecificData* points to. The **MaximumAllowed** member specifies the maximum amount of dump data the routine can provide.

The value of the **OutBuffer** member determines whether the system is requesting the size of the driver's dump data, or the data itself, as follows:

- If the **OutBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA is **NULL**, the system is only requesting size information. The *BugCheckSecondaryDumpDataCallback* routine fills in the **OutBuffer** and **OutBufferLength** members.
- If the **OutBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA equals the **InBuffer** member, the system is requesting the driver's secondary dump data. The *BugCheckSecondaryDumpDataCallback* routine fills in the **OutBuffer** and **OutBufferLength** members, and writes the data to the buffer specified by **OutBuffer**.

The **InBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA points to a small buffer for the routine's use. The **InBufferLength** member specifies the size of the buffer. If the amount of data to be written is less than **InBufferLength**, the callback routine can use this buffer to supply the crash dump data to the system. The callback routine then sets **OutBuffer** to **InBuffer** and **OutBufferLength** to the actual amount of data written to the buffer.

A driver that must write an amount of data that is larger than **InBufferLength** can use its own buffer to provide the data. This buffer must have been allocated before the callback routine is executed, and must reside in resident memory (such as nonpaged pool). The callback routine then sets **OutBuffer** to point to the driver's buffer, and **OutBufferLength** to the amount of data in the buffer to be written to the crash dump file.

In Windows XP and Windows Server 2003, if **OutBuffer** is set to point to a driver-allocated buffer, the buffer must begin on a page-aligned boundary in memory. Otherwise, no data is written to the secondary data area of the crash dump file. In Windows Vista and later versions of Windows, there is no such alignment requirement.

Each block of data to be written to the crash dump file is tagged with the value of the **Guid** member of KBUGCHECK_SECONDARY_DUMP_DATA. The GUID used must be unique to the driver. To display the secondary dump data corresponding to this GUID, you can use the **.enumtag** command or the **IDebugDataSpaces3::ReadTagged** method in a debugger extension. For information about debuggers and debugger extensions, see Windows Debugging.

A driver can write multiple blocks with the same GUID to the crash dump file, but this is very poor practice, because only the first block will be accessible to the debugger. Drivers that register multiple *BugCheckSecondaryDumpDataCallback* routines should allocate a unique GUID for each callback.

Use KeRegisterBugCheckReasonCallback to register a *BugCheckSecondaryDumpDataCallback* routine. A driver can subsequently remove the callback routine by using the KeDeregisterBugCheckReasonCallback routine. If the driver can be unloaded, then it must remove any registered callback routines in its Unload routine.

A *BugCheckSecondaryDumpDataCallback* is very restricted in the actions it can take. For more information, see Writing a Bug Check Callback Routine.

Examples

To define the callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *BugCheckAddPagesCallback* callback routine that is named `MyBugCheckAddPagesCallback`, use the KBUGCHECK_REASON_CALLBACK_ROUTINE type as shown in this code example:

```
KBUGCHECK_REASON_CALLBACK_ROUTINE MyBugCheckAddPagesCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyBugCheckAddPagesCallback(
    KBUGCHECK_CALLBACK_REASON  Reason,
    struct _KBUGCHECK_REASON_CALLBACK_RECORD  *Record,
    PVOID  ReasonSpecificData,
    ULONG  ReasonSpecificDataLength
    )
  {
      // Function body
  }
```

The KBUGCHECK_REASON_CALLBACK_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KBUGCHECK_REASON_CALLBACK_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows Server 2008.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at HIGH_LEVEL.

See Also

BugCheckSecondaryDumpDataCallback

KBUGCHECK_ADD_PAGES

KBUGCHECK_CALLBACK_REASON

KBUGCHECK_REASON_CALLBACK_RECORD

KeDeregisterBugCheckReasonCallback

KeRegisterBugCheckReasonCallback

KDEFERRED_ROUTINE callback function

1/25/2019 • 3 minutes to read • Edit Online

The callback routine performs actions, after an InterruptService returns, of a threaded DPC,

The *CustomDpc* routine finishes the servicing of an I/O operation, after an InterruptService routine returns.

The *CustomThreadedDpc* routine performs the action of a threaded DPC. The system executes this routine when the threaded DPC runs.

The *CustomTimerDpc* routine executes after a timer object's time interval expires.

Syntax

```
KDEFERRED_ROUTINE KdeferredRoutine;

void KdeferredRoutine(
  _KDPC *Dpc,
  PVOID DeferredContext,
  PVOID SystemArgument1,
  PVOID SystemArgument2
)
{...}
```

Parameters

`*Dpc`

Caller-supplied pointer to a KDPC structure, which represents the DPC object that is associated with this callback routine.

`DeferredContext`

For *CustomDpc*, a caller-supplied pointer to driver-defined context information that was specified in a previous call to KeInitializeDpc.

For *CustomThreadedDpc*, specifies driver-defined context information. When it initialized the DPC object, the driver supplied this value as the *DeferredContext* parameter to KeInitializeThreadedDpc.

Caller-supplied pointer to a KDPC structure, which represents the DPC object associated with this *CustomTimerDpc* routine.

`SystemArgument1`

Caller-supplied pointer to driver-supplied information that was specified in a previous call to KeInsertQueueDpc. When it added the DPC to the DPC queue, the driver supplied this value as the *SystemArgument1* parameter to **KeInsertQueueDpc**.

For *CustomTimerDpc*, this value is not used.

`SystemArgument2`

Specifies driver-defined context information. When it added the DPC to the DPC queue, the driver supplied this value as the *SystemArgument2* parameter to **KeInsertQueueDpc**.

For *CustomTimerDpc*, this value is not used.

Return Value

None

Remarks

About implementing *CustomDpc*: To create a DPC object and register a *CustomDpc* routine for that object, a driver must call KeInitializeDpc. (If you need only one DPC routine, you can use a DpcForIsr routine and the system-allocated DPC object.)

To queue a *CustomDpc* routine for execution, a driver's InterruptService routine must call **KeInsertQueueDpc**.

One or more *CustomDpc* routines can be used instead of, or in conjunction with, a *DpcForIsr* routine. A driver that maintains several internal IRP queues typically supplies a *CustomDpc* routine for each queue. Each *CustomDpc* routine is typically responsible for at least the following tasks:

- Completing the I/O operation that is described by the current IRP.
- Dequeuing the next IRP from one of the driver's IRP queues. (Drivers that use the system-supplied IRP queue together with a StartIo routine call IoStartNextPacket.)
- Setting the I/O status block in the current IRP and calling IoCompleteRequest for the completed request.

A *CustomDpc* routine might also retry a failed operation or set up the next transfer for a large I/O request that has been broken into smaller pieces.

For more information about *CustomDpc* routines, see DPC Objects and DPCs.

About implementing *CustomThreadedDpc*: A driver registers a *CustomThreadedDpc* for a DPC object by calling KeInitializeThreadedDpc. To actually add the DPC to the DPC queue so that the *CustomThreadedDpc* routine will be executed, call KeInsertQueueDpc.

For more information about using *CustomThreadedDpc* routines, see Threaded DPCs.

A *CustomThreadedDpc* routine can run at IRQL = DISPATCH_LEVEL, or it can run at IRQL = PASSIVE_LEVEL in a real-time thread.

About implementing *CustomTimerDpc*: To create a DPC object and register a *CustomTimerDpc* routine for that object, a driver must call KeInitializeDpc.

To queue a *CustomTimerDpc* routine for execution, a driver routine must call KeSetTimer or KeSetTimerEx, supplying a DPC object pointer returned by **KeInitializeDpc**. The system calls the *CustomTimerDpc* routine when the timer interval expires.

For more information about *CustomTimerDpc* routines, see Timer Objects and DPCs.

Examples

To define a callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *CustomDpc* callback routine that is named `MyCustomDpc`, use the KDEFERRED_ROUTINE type as shown in this code example:

```
```

```
KDEFERRED_ROUTINE MyCustomDpc;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyCustomDpc(
    struct _KDPC  *Dpc,
    PVOID  DeferredContext,
    PVOID  SystemArgument1,
    PVOID  SystemArgument2
    )
  {
      // Function body
  }
```

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DISPATCH_LEVEL.

See Also

KeInitializeDpc

KeInsertQueueDpc

KeSetTimer

KeSetTimerEx

KeAcquireSpinLock macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeAcquireSpinLock** routine acquires a spin lock so the caller can synchronize access to shared data in a multiprocessor-safe way by raising IRQL.

Syntax

```
void KeAcquireSpinLock(
   a,
   b
);
```

Parameters

`a`

Pointer to an initialized KSPIN_LOCK spin lock for which the caller provides the storage.

`b`

Pointer to a KIRQL variable that is set to the current IRQL when this call occurs.

Return Value

None

Remarks

KeAcquireSpinLock first resets the IRQL to DISPATCH_LEVEL and then acquires the lock. The previous IRQL is written to *OldIrql* after the lock is acquired.

The *OldIrql* value must be specified when the spin lock is released with KeReleaseSpinLock.

Most drivers use a local variable to store the old IRQL value. A driver can also use a shared memory location, such as a global variable, but the driver must not use the same location for two different locks. Otherwise, a race condition can occur.

Spin locks can cause serious problems if not used judiciously. In particular, no deadlock protection is performed and dispatching is disabled while the spin lock is held. Therefore:

- The code within a critical region guarded by an spin lock must neither be pageable nor make any references to pageable data.
- The code within a critical region guarded by a spin lock can neither call any external function that might access pageable data or raise an exception, nor can it generate any exceptions.
- The caller should release the spin lock with **KeReleaseSpinLock** as quickly as possible.

Attempting to acquire a spin lock recursively is guaranteed to cause a deadlock. For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, MarkingQueuedIrps, SpinLock, SpinLockDpc, SpinlockRelease, SpinLockSafe, ReqSendWhileSpinlock, Spinlock(kmdf), SpinlockDpc(kmdf), SpinlockRelease(kmdf), HwStorPortProhibitedDDIs, SpinLock(storport), SpinLockDpc(storport), SpinLockRelease(storport), SpinLockSafe(storport)

See Also

KeAcquireInStackQueuedSpinLock

KeAcquireSpinLockAtDpcLevel

KeInitializeSpinLock

KeReleaseSpinLock

KeAcquireSpinLockAtDpcLevel macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeAcquireSpinLockAtDpcLevel** routine acquires a spin lock when the caller is already running at IRQL >= DISPATCH_LEVEL.

Syntax

```
void KeAcquireSpinLockAtDpcLevel(
   a
);
```

Parameters

`a`

Pointer to an initialized KSPIN_LOCK spin lock for which the caller must provide the storage.

Return Value

None

Remarks

Drivers call **KeAcquireSpinLockAtDpcLevel** instead of KeAcquireSpinLock for better driver performance if and only if they are already running at an IRQL of DISPATCH_LEVEL or above.

If a driver is running at IRQL <= APC_LEVEL, it should call **KeAcquireSpinLock** to have IRQL raised by that routine. **KeAcquireSpinLockAtDpcLevel** assumes the caller is already running at IRQL >= DISPATCH_LEVEL, so no raise is necessary.

The caller should release the spin lock with KeReleaseSpinLockFromDpcLevel as quickly as possible.

For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

DDI compliance rules	HwStorPortProhibitedDDIs, IrqlDispatch, SpinLockSafe

See Also

KeAcquireInStackQueuedSpinLockAtDpcLevel

KeAcquireSpinLock

KeInitializeSpinLock

KeReleaseSpinLock

KeReleaseSpinLockFromDpcLevel

KeTryToAcquireSpinLockAtDpcLevel

KeAreAllApcsDisabled function

1/25/2019 • 2 minutes to read • Edit Online

The **KeAreAllApcsDisabled** routine indicates whether the calling thread is inside a guarded region or running at IRQL >= APC_LEVEL, which disables all APC delivery.

Syntax

```
BOOLEAN KeAreAllApcsDisabled(

);
```

Parameters

This function has no parameters.

Return Value

KeAreAllApcsDisabled returns **TRUE** if the calling thread is inside a guarded region or running at IRQL >= APC_LEVEL, and **FALSE** otherwise.

Remarks

A thread inside a guarded region or running at IRQL >= APC_LEVEL has all APCs disabled, including special kernel APCs.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAreApcsDisabled

KeAreApcsDisabled function

1/25/2019 • 2 minutes to read • Edit Online

The **KeAreApcsDisabled** routine returns whether the calling thread is within a critical region or a guarded region, which disables normal kernel APC delivery.

Syntax

```
BOOLEAN KeAreApcsDisabled(

);
```

Parameters

This function has no parameters.

Return Value

KeAreApcsDisabled returns **TRUE** if the thread is within a critical region or a guarded region, and **FALSE** otherwise.

Remarks

A thread running at IRQL = PASSIVE_LEVEL can use **KeAreApcsDisabled** to determine if normal kernel APCs are disabled. A thread that is inside a critical region has both user APCs and normal kernel APCs disabled, but not special kernel APCs. A thread that is inside a guarded region has all APCs disabled, including special kernel APCs.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAreAllApcsDisabled

KeEnterCriticalRegion

KeLeaveCriticalRegion

KeBugCheckEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeBugCheckEx** routine brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.

Syntax

```
void KeBugCheckEx(
  ULONG     BugCheckCode,
  ULONG_PTR BugCheckParameter1,
  ULONG_PTR BugCheckParameter2,
  ULONG_PTR BugCheckParameter3,
  ULONG_PTR BugCheckParameter4
);
```

Parameters

`BugCheckCode`

Specifies a value that indicates the reason for the bug check.

`BugCheckParameter1`

`BugCheckParameter2`

`BugCheckParameter3`

`BugCheckParameter4`

The four *BugCheckParameterX* values supply additional information, such as the address and data where a memory-corruption error occurred, depending on the value of *BugCheckCode*.

Return Value

None

Remarks

A bug check is a system-detected error that causes an immediate, controlled shutdown of the system. Various kernel-mode components perform run-time consistency checking. When such a component discovers an unrecoverable inconsistency, it causes a bug check to be generated.

Whenever possible, all kernel-mode components should log an error and continue to run, rather than calling **KeBugCheckEx**. For example, if a driver is unable to allocate required resources, it should log an error so that the system continues to run; it must not generate a bug check.

A driver or other kernel-mode component should call this routine only in cases of a fatal, unrecoverable error that could corrupt the system itself.

KeBugCheckEx can be useful in the early stages of developing a driver, or while it is undergoing testing. In these circumstances, the *BugCheckCode* passed to this routine should be distinct from those codes already in use by Windows or its drivers. For a list of these codes, see Bug Check Codes.

However, even during driver development, this routine is of only limited utility, since it results in a complete system shutdown. A more effective debugging method is to attach a kernel debugger to the system and then use routines that send messages to the debugger or break into the debugger. For further information, see Using Debugging Code in a Driver.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoAllocateErrorLogEntry

IoWriteErrorLogEntry

KeBugCheck

KeRegisterBugCheckCallback

KeRegisterBugCheckReasonCallback

KeCancelTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **KeCancelTimer** routine dequeues a timer object before the timer interval, if any was set, expires.

Syntax

```
BOOLEAN KeCancelTimer(
  PKTIMER Arg1
);
```

Parameters

`Arg1`

Pointer to an initialized timer object, for which the caller provides the storage.

Return Value

If the specified timer object is in the system timer queue, **KeCancelTimer** returns **TRUE**.

Remarks

If the timer object is currently in the system timer queue, it is removed from the queue. If a DPC object is associated with the timer, it too is canceled. Otherwise, no operation is performed.

The routine returns **TRUE** if the timer is still in the timer queue. A nonperiodic timer is removed from the system queue as soon as it expires. Thus, for nonperiodic timers, **KeCancelTimer** returns **FALSE** if the timer DPC has been queued. Periodic timers are always in the timer queue, so **KeCancelTimer** always returns **TRUE** for periodic timers.

Note that a DPC that is already running runs to completion. The driver must ensure that the DPC has completed before freeing any resources used by the DPC. For a nonperiodic timer, you can use synchronization primitives, such as event objects, to synchronize between the driver and the DPC. The driver can check the return code of **KeCancelTimer** to determine if the DPC is running. If so, the DPC can signal the event before exiting, and the driver can wait for that event to be reset to the not-signaled state.

Since for periodic timers **KeCancelTimer** always returns **TRUE**, drivers must use a different technique to wait until the DPC has completed. Use the KeFlushQueuedDpcs routine to block until the DPC executes.

Drivers do not need to synchronize for data stored in global variables or driver object extensions. The system automatically calls **KeFlushQueuedDpcs** before deallocating either of these regions.

For more information about timer objects, see Timer Objects and DPCs.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeInitializeTimer

KeReadStateTimer

KeSetTimer

KeClearEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KeClearEvent** routine sets an event to a not-signaled state.

Syntax

```
void KeClearEvent(
  PRKEVENT Event
);
```

Parameters

`Event`

Pointer to an initialized dispatcher object of type event for which the caller supplies the storage.

Return Value

None

Remarks

Event is set to a not-signaled state, meaning its value is set to zero.

For better performance, use **KeClearEvent** unless the caller uses the value returned by KeResetEvent to determine what to do next.

For more information about event objects, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IoBuildDeviceIoControlSetEvent, IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

KeReadStateEvent

KeResetEvent

KeSetEvent

KeConvertAuxiliaryCounterToPerformanceCounter function

1/25/2019 • 2 minutes to read • Edit Online

The **KeConvertAuxiliaryCounterToPerformanceCounter** routine converts the specified auxiliary counter value into a performance counter value.

Syntax

```
NTSTATUS KeConvertAuxiliaryCounterToPerformanceCounter(
  ULONG64  AuxiliaryCounterValue,
  PULONG64 PerformanceCounterValue,
  PULONG64 ConversionError
);
```

Parameters

`AuxiliaryCounterValue`

The auxiliary counter value to convert.

`PerformanceCounterValue`

A pointer to the variable that contains the converted performance counter value.

`ConversionError`

A pointer to a variable that contains the estimated conversion error in units of nanosecond.

Return Value

KeConvertAuxiliaryCounterToPerformanceCounter can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The conversion succeeded.
STATUS_NOT_SUPPORTED	Auxiliary counter is not supported.

STATUS_INVALID_PARAMETER	The *AuxiliaryCounterValue* value is not valid. For example, the value is earlier than the last system boot/recovery, or is out of the +/- 10s range compared to the current auxiliary counter value.
STATUS_UNSUCCESSFUL	The routine cannot convert the specified value with acceptable accuracy.

Remarks

Make sure that the specified auxiliary counter value is within +/- 10s compared to the current value.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Hal.dll
IRQL	Any level

See Also

KeConvertPerformanceCounterToAuxiliaryCounter

KeConvertPerformanceCounterToAuxiliaryCounter function

1/25/2019 • 2 minutes to read • Edit Online

The **KeConvertPerformanceCounterToAuxiliaryCounter** routine converts the specified performance counter value into an auxiliary counter value.

Syntax

```
NTSTATUS KeConvertPerformanceCounterToAuxiliaryCounter(
  ULONG64  PerformanceCounterValue,
  PULONG64 AuxiliaryCounterValue,
  PULONG64 ConversionError
);
```

Parameters

`PerformanceCounterValue`

The performance counter value to convert.

`AuxiliaryCounterValue`

A pointer to the variable that contains the converted auxiliary counter value.

`ConversionError`

A pointer to a variable that contains the estimated conversion error in units of nanosecond.

Return Value

KeConvertPerformanceCounterToAuxiliaryCounter can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The conversion succeeded.
STATUS_NOT_SUPPORTED	Auxiliary counter is not supported.

STATUS_INVALID_PARAMETER	The *PerformanceCounterValue* value is not valid. For example, the value is earlier than the last system boot/recovery, or is out of the +/- 10s range compared to the current performance counter value.
STATUS_UNSUCCESSFUL	The routine cannot convert the specified value with acceptable accuracy.

Remarks

Make sure that the specified performance counter value is:

- Within +/- 10s compared to the current performance counter read value.
- Not earlier than the recorded performance counter value at the last system boot or recovery from S3/S4 state.

The *ConversionError* value is the difference, in nanoseconds, between the expected calculated value and the actual calculated value for the auxiliary counter. If the *ConversionError* value is greater than the expected value (determined by you), then call the routine again.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Hal.dll
IRQL	Any level

See Also

KeConvertAuxiliaryCounterToPerformanceCounter

KeDelayExecutionThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDelayExecutionThread** routine puts the current thread into an alertable or nonalertable wait state for a specified interval.

Syntax

```
NTSTATUS KeDelayExecutionThread(
  KPROCESSOR_MODE WaitMode,
  BOOLEAN         Alertable,
  PLARGE_INTEGER  Interval
);
```

Parameters

`WaitMode`

Specifies the processor mode in which the caller is waiting, which can be either **KernelMode** or **UserMode**. Lower-level drivers should specify **KernelMode**.

`Alertable`

Specifies **TRUE** if the wait is alertable. Lower-level drivers should specify **FALSE**.

`Interval`

Specifies the absolute or relative time, in units of 100 nanoseconds, for which the wait is to occur. A negative value indicates relative time. Absolute expiration times track any changes in system time; relative expiration times are not affected by system time changes.

Return Value

KeDelayExecutionThread returns one of the following values that describes how the delay was completed:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The delay completed because the specified interval elapsed.
STATUS_ALERTED	The delay completed because the thread was alerted.

STA TUS _US ER_ APC	A user-mode APC was delivered before the specified *Interval* expired.

Note that the NT_SUCCESS macro recognizes all of these status values as "success" values.

Remarks

The expiration time is computed and the current thread is put in a wait state. When the specified interval has passed, the thread exits the wait state and is put in the ready state, becoming eligible for execution.

The *Alertable* parameter determines when the thread can be alerted and its wait state consequently aborted. For additional information, see Waits and APCs.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must never attempt to pass parameters on the stack when calling **KeDelayExecutionThread** using the **UserMode** argument.

It is especially important to check the return value of **KeDelayExecutionThread** when the *WaitMode* parameter is **UserMode** or *Alertable* is **TRUE**, because **KeDelayExecutionThread** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to **FALSE**.

Where possible, *Alertable* should be set to **FALSE** and *WaitMode* should be set to **KernelMode**, in order to reduce driver complexity. The principal exception to this guideline is when the wait is a long-term wait.

The expiration time of the delay is expressed as either an absolute time at which the delay is to expire, or a time relative to the current system time. If the *Interval* parameter value is positive, the expiration time is an absolute time. If this value is negative, the expiration time is a relative time.

Expiration times are measured relative to the system clock, and the accuracy with which the operating system can detect when a timer expires is limited by the granularity of the system clock. For more information, see Timer Accuracy.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

DDI compliance rules	IrqlKeApcLte1, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeQuerySystemTime

KeDeregisterBoundCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDeregisterBoundCallback** routine deregisters a user-mode bound exception callback registered by KeRegisterBoundCallback.

Syntax

```
NTSTATUS KeDeregisterBoundCallback(
  PVOID Handle
);
```

Parameters

`Handle`

Specifies the value returned by KeRegisterBoundCallback when the callback was registered.

Return Value

The **KeDeregisterBoundCallback** routine returns STATUS_SUCCESS if the callback is successfully removed. It returns STATUS_INVALID_HANDLE if no callback matching the provided *Handle* value is found.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

See Also

KeRegisterBoundCallback

KeDeregisterBugCheckCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDeregisterBugCheckCallback** routine removes a callback routine that was registered by KeRegisterBugCheckCallback.

Syntax

```
BOOLEAN KeDeregisterBugCheckCallback(
  PKBUGCHECK_CALLBACK_RECORD CallbackRecord
);
```

Parameters

`CallbackRecord`

Pointer to a KBUGCHECK_CALLBACK_RECORD structure. *CallbackRecord* must be the same value that was passed to KeRegisterBugCheckCallback when the callback was registered.

Return Value

KeDeregisterBugCheckCallback returns **TRUE** if the specified callback is successfully removed. It returns **FALSE** if the specified callback is not registered.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeRegisterBugCheckCallback

KeDeregisterBugCheckReasonCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDeregisterBugCheckReasonCallback** routine removes a callback routine that was registered by KeRegisterBugCheckReasonCallback.

Syntax

```
BOOLEAN KeDeregisterBugCheckReasonCallback(
  PKBUGCHECK_REASON_CALLBACK_RECORD CallbackRecord
);
```

Parameters

`CallbackRecord`

Pointer to a KBUGCHECK_REASON_CALLBACK_RECORD structure. *CallbackRecord* must be the same value that was passed to KeRegisterBugCheckReasonCallback when the callback was registered.

Return Value

KeDeregisterBugCheckReasonCallback returns **TRUE** if the callback is successfully removed. It returns **FALSE** if the specified callback is not registered.

Requirements

Minimum supported client	Available in Windows XP Service Pack 1 (SP1), Windows Server 2003, and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeRegisterBugCheckReasonCallback

KeDeregisterNmiCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDeregisterNmiCallback** routine deregisters a nonmaskable interrupt (NMI) callback registered by KeRegisterNmiCallback.

Syntax

```
NTSTATUS KeDeregisterNmiCallback(
  PVOID Handle
);
```

Parameters

`Handle`

Specifies the value returned by KeRegisterNmiCallback when the NMI callback was registered.

Return Value

The **KeDeregisterNmiCallback** routine returns STATUS_SUCCESS if the callback is successfully removed. It returns STATUS_INVALID_HANDLE if no callback matching the provided *Handle* value is found.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlKeApcLte2, HwStorPortProhibitedDDIs

See Also

KeRegisterNmiCallback

KeDeregisterProcessorChangeCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeDeregisterProcessorChangeCallback** routine unregisters a callback function that was previously registered with the operating system by calling the KeRegisterProcessorChangeCallback routine.

Syntax

```
void KeDeregisterProcessorChangeCallback(
  PVOID CallbackHandle
);
```

Parameters

`CallbackHandle`

The callback registration handle that was returned by the KeRegisterProcessorChangeCallback routine when the callback function was registered with the operating system.

Return Value

None

Remarks

A device driver calls the KeRegisterProcessorChangeCallback routine to register a callback function with the operating system so that the operating system will notify the driver when a new processor is added to the hardware partition. When the device driver no longer needs to receive notification when new processors are added to the hardware paritition, it calls the **KeDeregisterProcessorChangeCallback** routine to unregister the callback function. A callback function that has been registered for notification of processor changes must be unregistered before the device driver is unloaded from the operating system.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

KeRegisterProcessorChangeCallback

KeEnterCriticalRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeEnterCriticalRegion** routine temporarily disables the execution of normal kernel APCs, but does not prevent special kernel APCs from running.

Syntax

```
void KeEnterCriticalRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

A driver calls this routine to enter a critical region in which the execution of normal kernel APCs is deferred until this driver exits the critical region by calling the KeLeaveCriticalRegion routine. Any caller of **KeEnterCriticalRegion** should call **KeLeaveCriticalRegion** as quickly as possible after entering a critical region.

Highest-level drivers can call **KeEnterCriticalRegion** while running in the context of the thread that requested the current I/O operation.

A thread that is inside a critical region has both user APCs and normal kernel APCs disabled, but not special kernel APCs. For more information about these APC types, see Types of APCs.

Critical regions can be entered recursively and each call to **KeEnterCriticalRegion** must have a matching call to **KeLeaveCriticalRegion**.

A driver can use a critical region to acquire and release exclusive access to a shared resource. In this case, the ExEnterCriticalRegionAndAcquireResourceExclusive and ExReleaseResourceAndLeaveCriticalRegion routines can be used instead of the **KeEnterCriticalRegion** and **KeLeaveCriticalRegion** routines. For more information, see the code example in ExEnterCriticalRegionAndAcquireResourceExclusive.

For more information about APCs, see Asynchronous Procedure Calls.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	CriticalRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

ExEnterCriticalRegionAndAcquireResourceExclusive

ExReleaseResourceAndLeaveCriticalRegion

KeAreApcsDisabled

KeLeaveCriticalRegion

KeEnterGuardedRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeEnterGuardedRegion** routine enters a guarded region, which disables all kernel-mode APC delivery to the current thread.

Syntax

```
void KeEnterGuardedRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

To exit a guarded region entered with **KeEnterGuardedRegion**, call the **KeLeaveGuardedRegion** routine. Guarded regions can be nested. APCs are not reenabled until the thread exits the outermost guarded region.

For more information about guarded regions, see Critical Regions and Guarded Regions.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	GuardedRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs

See Also

KeLeaveGuardedRegion

KeFlushIoBuffers function

1/25/2019 • 2 minutes to read • Edit Online

The **KeFlushIoBuffers** routine flushes the memory region described by an MDL from caches of all processors.

Syntax

```
void KeFlushIoBuffers(
  PMDL    Mdl,
  BOOLEAN ReadOperation,
  BOOLEAN DmaOperation
);
```

Parameters

`Mdl`

A pointer to an MDL that describes the range for the I/O buffer.

`ReadOperation`

Specifies **TRUE** if the flush is being performed for a read operation, and **FALSE** for a write operation. For more information, see the following Remarks section.

`DmaOperation`

Specifies **TRUE** for a DMA transfer, **FALSE** for PIO.

Return Value

None

Remarks

Drivers call **KeFlushIoBuffers** to maintain data integrity during DMA or PIO device transfer operations. Calling this routine affects all processors in the machine.

If *ReadOperation* is **TRUE**, the driver is reading information from the device to system memory, so valid data still might be in the processor instruction and data caches. **KeFlushIoBuffers** flushes data from all processors' caches to system memory, including the processor on which the caller is running.

If *ReadOperation* is **FALSE**, the driver is writing data from system memory to a device, so valid data might be in the processor's data cache but not yet transferred to the device. **KeFlushIoBuffers** flushes all processors' data caches, including that of the processor on which the caller is running.

As a general rule, drivers should call **KeFlushIoBuffers** just before beginning a DMA transfer operation or immediately following any PIO read operation.

KeFlushIoBuffers flushes only the memory region that is described by the MDL that the *Mdl* parameter points to. To flush all the memory regions that are described by an MDL chain requires a separate **KeFlushIoBuffers** call for each MDL in the chain.

For more information, see the following topics:

Flushing Cached Data during DMA Operations Flushing Cached Data during PIO Operations

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	MdlAfterReqCompletedIntIoctlA, MdlAfterReqCompletedIoctlA, MdlAfterReqCompletedReadA, MdlAfterReqCompletedWriteA

See Also

FlushAdapterBuffers

KeFlushQueuedDpcs function

1/25/2019 • 2 minutes to read • Edit Online

The **KeFlushQueuedDpcs** routine returns after all queued DPCs on all processors have executed.

Syntax

```
void KeFlushQueuedDpcs(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

Drivers can use this routine to wait until all currently-queued DPCs are run. Note that **KeFlushQueuedDpcs** can take a long time to return, so drivers should not use it along any critical code paths.

Requirements

Minimum supported client	Available starting with Windows XP with SP2 and Windows Server 2003.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoInitializeDpcRequest

IoRequestDpc

KeInitializeDpc

KeInsertQueueDpc

KefReleaseSpinLockFromDpcLevel function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseSpinLockFromDpcLevel** routine releases an executive spin lock without changing the IRQL.

Syntax

```
void KefReleaseSpinLockFromDpcLevel(
  PKSPIN_LOCK SpinLock
);
```

Parameters

`SpinLock`

Pointer to an executive spin lock for which the caller provides the storage.

Return Value

None

Remarks

Drivers call **KeReleaseSpinLockFromDpcLevel** to release a spin lock acquired by calling KeAcquireSpinLockAtDpcLevel.

It is an error to call **KeReleaseSpinLockFromDpcLevel** if the specified spin lock was acquired by calling KeAcquireSpinLock because the caller's original IRQL is not restored, which can cause deadlocks or fatal page faults.

For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs, IrqlDispatch, SpinLockSafe

See Also

KeAcquireSpinLock

KeAcquireSpinLockAtDpcLevel

KeReleaseSpinLock

KeGetCurrentIrql function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentIrql** routine returns the current IRQL. For information about IRQLs, see Managing Hardware Priorities.

Syntax

```
NTHALAPI KIRQL KeGetCurrentIrql(

);
```

Parameters

This function has no parameters.

Return Value

KeGetCurrentIrql returns the current IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
IRQL	Any level
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAcquireSpinLockAtDpcLevel

KeLowerIrql

KeRaiseIrql

Managing Hardware Priorities

KeGetCurrentNodeNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentNodeNumber** routine gets the NUMA node number for the logical processor that the caller is running on.

Syntax

```
USHORT KeGetCurrentNodeNumber(

);
```

Parameters

This function has no parameters.

Return Value

KeGetCurrentNodeNumber returns the node number.

Remarks

In a non-uniform memory access (NUMA) multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

In a NUMA multiprocessor system that contains *n* nodes, the nodes are numbered from 0 to *n*-1. To get the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryHighestNodeNumber

KeGetCurrentProcessorNumberEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentProcessorNumberEx** routine gets the processor number of the logical processor that the caller is running on.

Syntax

```
ULONG KeGetCurrentProcessorNumberEx(
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcNumber`

A pointer to a caller-allocated buffer into which the routine writes a PROCESSOR_NUMBER structure that identifies the current logical processor. This structure contains the group number and the group-relative processor number. You can set this parameter to **NULL** if you do not need this information.

Return Value

KeGetCurrentProcessorNumberEx returns the systemwide processor index of the logical processor that the caller is running on.

Remarks

If *ProcNumber* is non-**NULL**, the buffer that is pointed to by *ProcNumber* must be large enough to contain a PROCESSOR_NUMBER structure. In contrast to this structure, which contains a group number and a group-relative processor number, the return value is a processor index that identifies the processor across the entire multiprocessor system.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

A related routine, KeGetCurrentProcessorNumber, returns the current processor number, but this routine, unlike **KeGetCurrentProcessorNumberEx**, does not provide a group number for the processor. In Windows 7 and later versions of the Windows operating system, **KeGetCurrentProcessorNumber** returns the group-relative processor number if the caller is running on a processor in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. However, if the caller is running on a processor in any group other than group 0, this routine returns a number that is less than the number of processors in group 0. This behavior ensures that the return value is less than the return value of the KeQueryActiveProcessorCount routine. Existing drivers that call **KeGetCurrentProcessorNumber** and that use no group-oriented features run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeGetCurrentProcessorNumberEx** instead of **KeGetCurrentProcessorNumber**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentProcessorNumber

KeQueryActiveProcessorCount

PROCESSOR_NUMBER

KeGetCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetCurrentThread** routine identifies the current thread.

Syntax

```
PKTHREAD KeGetCurrentThread(

);
```

Parameters

This function has no parameters.

Return Value

KeGetCurrentThread returns a pointer to an opaque thread object.

Remarks

This routine is identical to PsGetCurrentThread.

A caller of **KeGetCurrentThread** can use the returned pointer as an input parameter to KeQueryPriorityThread, KeSetBasePriorityThread, or KeSetPriorityThread. However, the memory containing the thread object is opaque; that is, it is reserved for exclusive use by the operating system.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryPriorityThread

KeSetBasePriorityThread

KeSetPriorityThread

PsGetCurrentThread

KeGetProcessorIndexFromNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetProcessorIndexFromNumber** routine converts a group number and a group-relative processor number to a systemwide processor index.

Syntax

```
ULONG KeGetProcessorIndexFromNumber(
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcNumber`

A pointer to a caller-allocated PROCESSOR_NUMBER structure that contains a group number and a group-relative processor number.

Return Value

KeGetProcessorIndexFromNumber returns a systemwide processor index if the call is successful. If *ProcNumber* points to an invalid **PROCESSOR_NUMBER** value, the routine returns INVALID_PROCESSOR_INDEX, which is defined in the Wdm.h header file.

Remarks

This routine accepts as input a **PROCESSOR_NUMBER** structure that identifies a processor by its group number and its processor number within the group. The return value is a processor index that identifies the processor across the entire multiprocessor system.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

To obtain the total number of active logical processors in the system, call the KeQueryActiveProcessorCountEx routine and set this routine's *GroupNumber* parameter to ALL_PROCESSOR_GROUPS.

The KeGetProcessorNumberFromIndex routine converts a systemwide processor index to a group number and a group-relative processor number.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorNumberFromIndex

KeQueryActiveProcessorCountEx

PROCESSOR_NUMBER

KeGetProcessorNumberFromIndex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetProcessorNumberFromIndex** routine converts a systemwide processor index to a group number and a group-relative processor number.

Syntax

```
NTSTATUS KeGetProcessorNumberFromIndex(
  ULONG             ProcIndex,
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`ProcIndex`

A systemwide processor index. If a multiprocessor system contains a total of *n* logical processors, valid processor indexes range from 0 to *n*-1.

`ProcNumber`

A pointer to a caller-allocated PROCESSOR_NUMBER structure into which the routine writes the group number and group-relative processor number of the processor that is identified by *ProcIndex*.

Return Value

KeGetProcessorNumberFromIndex returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ProcIndex* parameter value is not a valid processor index.

Remarks

This routine accepts as input a processor index that identifies the processor across the entire multiprocessor system. The output value is a **PROCESSOR_NUMBER** structure that identifies a processor by its group number and its processor number within the group.

For example, if a multiprocessor system contains two groups, and each group contains 64 logical processors, the processor numbers in each group range from 0 to 63, but the systemwide processor indexes range from 0 to 127.

To obtain the total number of active logical processors in the system, call the KeQueryActiveProcessorCountEx routine.

The KeGetProcessorIndexFromNumber routine converts a group number and a group-relative processor number to a systemwide processor index.

Examples

The following code example uses the **KeQueryActiveProcessorCountEx** and **KeGetProcessorNumberFromIndex** routines to enumerate all active logical processors in the system:

```
ULONG Count;
ULONG ProcIndex;
PROCESSOR_NUMBER ProcNumber;

Count = KeQueryActiveProcessorCountEx(ALL_PROCESSOR_GROUPS);
for (ProcIndex = 0; ProcIndex < Count; ProcIndex += 1)
{
    KeGetProcessorNumberFromIndex(ProcIndex, &ProcNumber);

    // Do something with the contents of ProcNumber.
    ...
}
```

The constant value ALL_PROCESSOR_GROUPS is defined in Winnt.h and Ntdef.h.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorIndexFromNumber

KeQueryActiveProcessorCountEx

PROCESSOR_NUMBER

KeGetRecommendedSharedDataAlignment function

1/25/2019 • 2 minutes to read • Edit Online

The **KeGetRecommendedSharedDataAlignment** routine returns the preferred alignment for memory structures that can be accessed by more than one processor.

Syntax

```
ULONG KeGetRecommendedSharedDataAlignment(

);
```

Parameters

This function has no parameters.

Return Value

KeGetRecommendedSharedDataAlignment returns the preferred alignment, in bytes, for memory structures that can be shared by more than one processor.

Remarks

Use **KeGetRecommendedSharedDataAlignment** to determine the best alignment for data structures that will be shared between processors. The value returned minimizes cache effects that negatively impact performance on multiprocessor systems.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

KeInitializeCrashDumpHeader function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeCrashDumpHeader** routine supplies the header information the system requires for a crash dump file.

Syntax

```
NTSTATUS KeInitializeCrashDumpHeader(
  ULONG  DumpType,
  ULONG  Flags,
  PVOID  Buffer,
  ULONG  BufferSize,
  PULONG BufferNeeded
);
```

Parameters

`DumpType`

Specifies the type of dump file. The only valid value is DUMP_TYPE_FULL.

`Flags`

Specifies flags for the dump file. The only valid value is 0.

`Buffer`

Pointer to the buffer that receives the header information.

`BufferSize`

Specifies the size in bytes of the buffer pointed to by *Buffer*.

`BufferNeeded`

Optionally, a pointer to a variable that receives the size necessary to hold the complete header information.

Return Value

KeInitializeCrashDumpHeader returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

Drivers can use this routine to manually create a crash dump file. The file can be created at any time, and used by a debugger to examine the state of the system.

To create a crash-dump file, call **KeInitializeCrashDumpHeader** to create the header, then append the contents of memory to the header. Note that the driver is not required to record the contents of memory immediately after calling the routine: the header can normally be created at any time before the crash-dump file is written.

Because the routine is designed to be called well before the memory contents are recorded, it has the following limitations:

- The routine does not record any information about active exception records.
- If the size of system RAM changes, the header must be recreated.

The routine does not record any secondary dump data.

Starting with Windows 8, **KeInitializeCrashDumpHeader** always writes the base address of the system process page directory to the crash-dump header, regardless of the process context from which **KeInitializeCrashDumpHeader** is called. The debugger can use this directory to access the crash-dump file in the context of the system process.

In earlier versions of Windows, **KeInitializeCrashDumpHeader** writes the base address of the page directory of the caller's current process context to the crash-dump header. Thus, **KeInitializeCrashDumpHeader** must be called from the system process. Otherwise, the debugger will be unable to access the crash-dump file in the process context in which the file was saved.

Starting with Windows 8, **KeInitializeCrashDumpHeader** is declared in the Wdm.h header file in the Windows Driver Kit (WDK). To use this routine with earlier versions of the WDK, include the following function declaration in your driver code:

C++

```
NTSTATUS
  KeInitializeCrashDumpHeader(
    __in ULONG  DumpType,
    __in ULONG  Flags,
    __out PVOID  Buffer,
    __in ULONG  BufferSize,
    __out_opt PULONG  BufferNeeded
    );
```

Requirements

Minimum supported client	Available starting with Windows Server 2003 with SP1.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

KeInitializeDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeDeviceQueue** routine initializes a device queue object to a not-busy state.

Syntax

```
void KeInitializeDeviceQueue(
  PKDEVICE_QUEUE DeviceQueue
);
```

Parameters

`DeviceQueue`

Pointer to a device queue object for which the caller provides the storage.

Return Value

None

Remarks

KeInitializeDeviceQueue initializes the specified device queue and sets its state to not-busy.

A driver should call **KeInitializeDeviceQueue** from its *AddDevice* routine after creating the device object for the associated device. Storage for the device queue object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeInsertByKeyDeviceQueue

KeInsertDeviceQueue

KeRemoveDeviceQueue

KeRemoveEntryDeviceQueue

KeInitializeDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeDpc** routine initializes a DPC object, and registers a CustomDpc routine for that object.

Syntax

```
void KeInitializeDpc(
  __drv_aliasesMem PRKDPC Dpc,
  PKDEFERRED_ROUTINE      DeferredRoutine,
  __drv_aliasesMem PVOID  DeferredContext
);
```

Parameters

`Dpc`

Pointer to a KDPC structure that represents the DPC object to initialize. The caller must allocate storage for the structure from resident memory.

`DeferredRoutine`

Pointer to the CustomDpc routine to associate with the DPC.

`DeferredContext`

Specifies the value to pass as the *DeferredContext* parameter to CustomDpc.

Return Value

None

Remarks

The caller can queue an initialized DPC with KeInsertQueueDpc. The caller also can set up a timer object associated with the initialized DPC object and queue the DPC with KeSetTimer.

Storage for the DPC object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level

See Also

CustomDpc

KeInsertQueueDpc

KeRemoveQueueDpc

KeSetTimer

KeInitializeEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeEvent** routine initializes an event object as a synchronization (single waiter) or notification type event and sets it to a signaled or not-signaled state.

Syntax

```
void KeInitializeEvent(
  PRKEVENT   Event,
  EVENT_TYPE Type,
  BOOLEAN    State
);
```

Parameters

`Event`

Pointer to an event object, for which the caller provides the storage.

`Type`

Specifies the event type, either **NotificationEvent** or **SynchronizationEvent**.

`State`

Specifies the initial state of the event. **TRUE** indicates a signaled state.

Return Value

None

Remarks

Storage for an event object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller. If you allocate the event on the stack, you must specify a **KernelMode** wait when calling KeWaitForSingleObject, KeWaitForMutexObject, or KeWaitForMultipleObjects. During a **KernelMode** wait, the stack containing the event will not be paged out.

Drivers typically use a **NotificationEvent** to wait for an I/O operation to complete. When a notification event is set to the signaled state, all threads that were waiting for the event to be set to the signaled state become eligible for execution. The event remains in the signaled state until a thread calls KeResetEvent or KeClearEvent to set the event in the not-signaled state.

A **SynchronizationEvent** is also called an *autoreset* or *autoclearing* event. When such an event is set, a single waiting thread becomes eligible for execution. The kernel automatically resets the event to the not-signaled state each time a wait is satisfied. A driver might use a synchronization event to protect a shared resource that is used in synchronizing the operations of several threads. Synchronization events are rarely used in a typical driver.

For more information about event objects, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level
DDI compliance rules	IoAllocateIrpSignalEventInCompletion, IoAllocateIrpSignalEventInCompletion2, IoAllocateIrpSignalEventInCompletion3, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildDeviceIoControlSetEvent, IoBuildFsdIrpSignalEventInCompletion, IoBuildFsdIrpSignalEventInCompletion2, IoBuildFsdIrpSignalEventInCompletion3, IoBuildFsdIrpSignalEventInCompletionTimeout, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, PendedCompletedRequest, PendedCompletedRequestEx, SignalEventInCompletion, SignalEventInCompletion2, SignalEventInCompletion3, StartDeviceWait, StartDeviceWait2, HwStorPortProhibitedDDIs

See Also

KeClearEvent

KeReadStateEvent

KeResetEvent

KeSetEvent

KeWaitForMultipleObjects

KeWaitForSingleObject

KeInitializeGuardedMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeGuardedMutex** routine initializes a guarded mutex.

Syntax

```
void KeInitializeGuardedMutex(
  PKGUARDED_MUTEX Mutex
);
```

Parameters

`Mutex`

Pointer to the KGUARDED_MUTEX structure to initialize. The storage must be 4-byte aligned on 32-bit platforms, and 8-byte aligned on 64-bit platforms.

Return Value

None

Remarks

Guarded mutexes must be allocated from nonpaged pool.

For more information about guarded mutexes, see Fast Mutexes and Guarded Mutexes.

Requirements

Minimum supported client	Available in Windows Server 2003 with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KGUARDED_MUTEX

KeAcquireGuardedMutex

KeInitializeMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeMutex** routine initializes a mutex object, setting it to a signaled state.

Syntax

```
void KeInitializeMutex(
  PRKMUTEX Mutex,
  ULONG    Level
);
```

Parameters

`Mutex`

Pointer to a mutex object, for which the caller provides the storage. The storage must be 4-byte aligned on 32-bit platforms, and 8-byte aligned on 64-bit platforms.

`Level`

Reserved. Drivers set this to zero.

Return Value

None

Remarks

For better performance, use fast mutexes or guarded mutexes. For more information, see Alternatives to Mutex Objects.

The mutex object is initialized with an initial state of signaled.

Storage for a mutex object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

For more information about mutex objects, see Mutex Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExInitializeFastMutex

KeReadStateMutex

KeReleaseMutex

KeWaitForMultipleObjects

KeWaitForMutexObject

KeWaitForSingleObject

KeInitializeSemaphore function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeSemaphore** routine initializes a semaphore object with a specified count and specifies an upper limit that the count can attain.

Syntax

```
void KeInitializeSemaphore(
  PRKSEMAPHORE Semaphore,
  LONG         Count,
  LONG         Limit
);
```

Parameters

`Semaphore`

Pointer to a dispatcher object of type semaphore, for which the caller provides the storage.

`Count`

Specifies the initial count value to be assigned to the semaphore. This value must be positive. A nonzero value sets the initial state of the semaphore to signaled.

`Limit`

Specifies the maximum count value that the semaphore can attain. This value must be positive. It determines how many waiting threads become eligible for execution when the semaphore is set to the signaled state and can therefore access the resource that the semaphore protects.

Return Value

None

Remarks

The semaphore object is initialized with the specified initial count and limit.

Storage for a semaphore object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

For more information about semaphore objects, see Semaphore Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlKeDispatchLte, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeReadStateSemaphore

KeReleaseSemaphore

KeWaitForMultipleObjects

KeWaitForSingleObject

KeInitializeSpinLock function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeSpinLock** routine initializes a variable of type KSPIN_LOCK.

Syntax

```
void KeInitializeSpinLock(
  PKSPIN_LOCK SpinLock
);
```

Parameters

`SpinLock`

Pointer to a spin lock, for which the caller must provide the storage.

Return Value

None

Remarks

This routine must be called before an initial call to KeAcquireSpinLock, to KeAcquireInStackQueuedSpinLock, or to any other support routine that requires a spin lock as an argument.

Storage for a spin lock object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

For more information about spin locks, see Spin Locks.

Callers of this routine can be running at any IRQL. Usually, a caller is running at IRQL = PASSIVE_LEVEL in an AddDevice routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAcquireInStackQueuedSpinLock

KeAcquireInStackQueuedSpinLockAtDpcLevel

KeAcquireSpinLock

KeAcquireSpinLockAtDpcLevel

KeReleaseInStackQueuedSpinLock

KeReleaseInStackQueuedSpinLockFromDpcLevel

KeReleaseSpinLock

KeReleaseSpinLockFromDpcLevel

KeInitializeThreadedDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeThreadedDpc** routine initializes a threaded DPC object, and registers a CustomThreadedDpc routine for that object.

Syntax

```
void KeInitializeThreadedDpc(
  PRKDPC             Dpc,
  PKDEFERRED_ROUTINE DeferredRoutine,
  PVOID              DeferredContext
);
```

Parameters

`Dpc`

Pointer to a KDPC structure that represents the DPC object to initialize. The caller must allocate storage for the structure from resident memory.

`DeferredRoutine`

Pointer to the CustomThreadedDpc routine to associate with the DPC.

`DeferredContext`

Specifies the value to pass as the *DeferredContext* parameter to CustomThreadedDpc.

Return Value

None

Remarks

For more information about threaded DPCs, see Threaded DPCs.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level

See Also

CustomThreadedDpc

KeInsertQueueDpc

KeRemoveQueueDpc

KeInitializeTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeTimer** routine initializes a timer object.

Syntax

```
void KeInitializeTimer(
  PKTIMER Timer
);
```

Parameters

`Timer`

Pointer to a timer object, for which the caller provides the storage.

Return Value

None

Remarks

The timer object is initialized to a not-signaled state.

Storage for a timer object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

KeInitializeTimer can only initialize a notification timer. Use KeInitializeTimerEx to initialize a notification timer or a synchronization timer.

Use KeSetTimer or KeSetTimerEx to define when the timer will expire.

For more information about timer objects, see Timer Objects and DPCs.

Callers of **KeInitializeTimer** should be running at IRQL = DISPATCH_LEVEL or lower. It is best to initialize timers at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeCancelTimer

KeInitializeTimerEx

KeReadStateTimer

KeSetTimer

KeSetTimerEx

KeWaitForMultipleObjects

KeWaitForSingleObject

KeInitializeTimerEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInitializeTimerEx** routine initializes an extended kernel timer object.

Syntax

```
void KeInitializeTimerEx(
  PKTIMER    Timer,
  TIMER_TYPE Type
);
```

Parameters

`Timer`

Pointer to a timer object, for which the caller provides the storage.

`Type`

Specifies the type of the timer object, either **NotificationTimer** or **SynchronizationTimer**.

Return Value

None

Remarks

The timer object is initialized to a not-signaled state.

Storage for a timer object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

When a notification timer expires, all waiting threads are released and the timer remains in the signaled state until it is explicitly reset. When a synchronization timer expires, it is set to a signaled state until a single waiting thread is released and then the timer is reset to a not-signaled state.

Callers of **KeInitializeTimerEx** should be running at IRQL = DISPATCH_LEVEL or lower. It is best to initialize timers at IRQL = PASSIVE_LEVEL.

For more information about timer objects, see Timer Objects and DPCs.

Use KeSetTimer or KeSetTimerEx to define when the timer will expire.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeCancelTimer

KeReadStateTimer

KeSetTimer

KeSetTimerEx

KeWaitForMultipleObjects

KeWaitForSingleObject

KeInsertByKeyDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInsertByKeyDeviceQueue** routine acquires the spin lock for the specified *DeviceQueue* and queues an entry according to the specified sort-key value if the device queue is set to a busy state.

Syntax

```
BOOLEAN KeInsertByKeyDeviceQueue(
  PKDEVICE_QUEUE       DeviceQueue,
  PKDEVICE_QUEUE_ENTRY DeviceQueueEntry,
  ULONG                SortKey
);
```

Parameters

`DeviceQueue`

Pointer to a control object of the device queue type for which the caller provides the storage.

`DeviceQueueEntry`

Pointer to the device queue entry to be inserted into the device queue according to the specific key value.

`SortKey`

Specifies the sort-key value that determines the position in the device queue in which to insert the entry.

Return Value

If the device queue is empty, **FALSE** is returned, meaning the *DeviceQueueEntry* is not inserted in the device queue.

Remarks

The specified device queue spin lock is acquired and the state of the device queue is checked. If the device queue is set to a busy state, the IRP specified by the *DeviceQueueEntry* is inserted into the device queue according to its sort key value and the device queue spin lock is released.

The new entry is positioned in the device queue after any entries in the queue with sort key values less than or equal to its sort key value and preceding any entries with sort key values that are greater.

If **KeInsertByKeyDeviceQueue** returns **FALSE**, the caller must begin processing the IRP. A call to KeInsertDeviceQueue or **KeInsertByKeyDeviceQueue** when the queue is empty causes the device queue to transition from a not-busy state to a busy state.

This routine is for code that queues an I/O request to a device driver.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, MarkingQueuedIrps, HwStorPortProhibitedDDIs, IrqlDispatch(storport)

See Also

KeInitializeDeviceQueue

KeInsertDeviceQueue

KeRemoveDeviceQueue

KeRemoveEntryDeviceQueue

KeInsertDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInsertDeviceQueue** routine acquires the spin lock for the specified device queue object and, if the device queue is set to a busy state, queues the specified entry.

Syntax

```
BOOLEAN KeInsertDeviceQueue(
  PKDEVICE_QUEUE       DeviceQueue,
  PKDEVICE_QUEUE_ENTRY DeviceQueueEntry
);
```

Parameters

`DeviceQueue`

Pointer to a control object of type device queue for which the caller provides the storage.

`DeviceQueueEntry`

Pointer to the device queue entry that is to be inserted.

Return Value

If the device queue is empty, **FALSE** is returned and the *DeviceQueueEntry* is not inserted in the device queue.

Remarks

If the device queue is set to a busy state, the specified *DeviceQueueEntry* is inserted at the tail of the device queue and the device queue spin lock is released.

If **KeInsertDeviceQueue** returns **FALSE**, the entry was not queued and the caller must begin processing the IRP. A call to **KeInsertDeviceQueue** or KeInsertByKeyDeviceQueue when the queue is empty causes the device queue to change from a not-busy state to a busy state.

This routine is for code that queues an I/O request to a device driver.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, MarkingQueuedIrps, HwStorPortProhibitedDDIs, IrqlDispatch(storport)

See Also

KeInitializeDeviceQueue

KeInsertByKeyDeviceQueue

KeRemoveDeviceQueue

KeRemoveEntryDeviceQueue

KeInsertQueueDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeInsertQueueDpc** routine queues a DPC for execution.

Syntax

```
BOOLEAN KeInsertQueueDpc(
  PRKDPC                 Dpc,
  PVOID                  SystemArgument1,
  __drv_aliasesMem PVOID SystemArgument2
);
```

Parameters

`Dpc`

Pointer to the KDPC structure for the DPC object. This structure must have been initialized by either KeInitializeDpc or KeInitializeThreadedDpc.

`SystemArgument1`

Specifies driver-determined context data. This value is passed as the *SystemArgument1* parameter to the DPC object's CustomDpc or CustomThreadedDpc routine.

`SystemArgument2`

Specifies driver-determined context data. This value is passed as the *SystemArgument2* parameter to the DPC object's CustomDpc or CustomThreadedDpc routine.

Return Value

If the specified DPC object is not currently in a DPC queue, **KeInsertQueueDpc** queues the DPC and returns **TRUE**.

Remarks

If the specified DPC object has already been queued, no operation is performed except to return **FALSE**. Otherwise, the DPC object is inserted in a DPC queue. For more information about DPC queues, see Organization of DPC Queues.

Note that a particular DPC object and the function that it represents can each be queued for execution only once at any particular time.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level
DDI compliance rules	MarkingQueuedIrps, HwStorPortProhibitedDDIs

See Also

CustomDpc

CustomThreadedDpc

KeInitializeDpc

KeRemoveQueueDpc

KeIpiGenericCall function

1/25/2019 • 2 minutes to read • Edit Online

The **KeIpiGenericCall** routine causes the specified routine to run on all processors simultaneously.

Syntax

```
ULONG_PTR KeIpiGenericCall(
  PKIPI_BROADCAST_WORKER BroadcastFunction,
  ULONG_PTR              Context
);
```

Parameters

`BroadcastFunction`

Pointer to an IpiGenericCall routine. This routine is run on every processor simultaneously.

`Context`

Specifies the value to pass to *IpiGenericCall* when it is called.

Return Value

KeIpiGenericCall returns the value that IpiGenericCall returns on the source processor (the processor that called **KeIpiGenericCall**).

Remarks

When a driver calls **KeIpiGenericCall**, the system interrupts every processor and raises the IRQL to IPI_LEVEL (interprocessor interrupt level). Each processor spins on a barrier until all processors have reached the barrier; then, all processors begin calling IpiGenericCall. **KeIpiGenericCall** waits for all calls to *IpiGenericCall* to complete before returning.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"< IPI_LEVEL"

See Also

IpiGenericCall

KeLeaveCriticalRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeLeaveCriticalRegion** routine reenables the delivery of normal kernel-mode APCs that were disabled by a preceding call to **KeEnterCriticalRegion**.

Syntax

```
void KeLeaveCriticalRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

Highest-level drivers can call this routine while running in the context of the thread that requested the current I/O operation.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	CriticalRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs, WithinCriticalRegion(storport)

See Also

KeAreApcsDisabled

KeEnterCriticalRegion

KeLeaveGuardedRegion function

1/25/2019 • 2 minutes to read • Edit Online

The **KeLeaveGuardedRegion** routine exits a guarded region entered by **KeEnterGuardedRegion**.

Syntax

```
void KeLeaveGuardedRegion(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

Guarded regions can be nested, so APCs are only reenabled once the thread exists the outermost guarded region.

For more information about guarded regions, see Critical Regions and Guarded Regions.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	GuardedRegions, IrqlKeApcLte2, WithinCriticalRegion, HwStorPortProhibitedDDIs

See Also

KeEnterGuardedRegion

KeLowerIrql function

1/25/2019 • 2 minutes to read • Edit Online

The **KeLowerIrql** routine restores the IRQL on the current processor to its original value. For information about IRQLs, see Managing Hardware Priorities.

Syntax

```
void KeLowerIrql(
  KIRQL NewIrql
);
```

Parameters

`NewIrql`

[in] Specifies the IRQL that was returned from KeRaiseIrql or KeRaiseIrqlToDpcLevel.

Return Value

None

Remarks

It is a fatal error to call **KeLowerIrql** using an input *NewIrql* that was not returned by the immediately preceding call to **KeRaiseIrql** or KeRaiseIrqlToDpcLevel.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
IRQL	Any level
DDI compliance rules	IrqlKeRaiseLower, IrqlKeRaiseLower2, HwStorPortProhibitedDDIs

See Also

KeGetCurrentIrql

KeRaiseIrql

KeRaiseIrqlToDpcLevel

KeMemoryBarrier function

1/25/2019 • 2 minutes to read • Edit Online

The **KeMemoryBarrier** routine creates a barrier at its position in the code—across which the compiler and the processor cannot move any operations.

Syntax

```
void KeMemoryBarrier(

);
```

Parameters

This function has no parameters.

Return Value

None

Remarks

The **KeMemoryBarrier** routine inserts a memory barrier into your code. This barrier guarantees that every operation that appears in the source code before the call to **KeMemoryBarrier** will complete before any operation that appears after the call.

The implementation of the **KeMemoryBarrier** routine depends on the processor architecture. For example, for an x86 processor, the Wdm.h header file defines **KeMemoryBarrier** to be the following inline function:

```
FORCEINLINE
VOID
KeMemoryBarrier (
    VOID
    )
{
    LONG Barrier;

    __asm {
        xchg Barrier, eax
    }
}
```

In this definition, the braces that follow the **__asm** keyword contain inline assembly code. The compiler optimizer cannot move an instruction from a position before the inline assembly code to a position after the inline assembly code, and vice versa. In addition, the **xchg** instruction implicitly includes the **lock** prefix, which forces the processor hardware to complete the memory operations for all instructions that precede the **xchg** instruction before it initiates memory operations for instructions that follow the **xchg** instruction.

KeMemoryBarrier prevents both the compiler and the processor from moving operations across the barrier. To

prevent only the compiler from moving operations, call KeMemoryBarrierWithoutFence.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level

See Also

KeMemoryBarrierWithoutFence

KeQueryActiveGroupCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveGroupCount** routine returns the number of active processor groups in a multiprocessor system.

Syntax

```
USHORT KeQueryActiveGroupCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryActiveGroupCount returns a count of the active groups.

Remarks

A processor group is active if it contains one or more active logical processors.

An active logical processor is a logical processor that Windows has started up and added to a multiprocessor system. The term *active processor* applies to a processor that is available to perform processing work, regardless of whether the processor is currently performing processing work or is idle. In some systems, the number of active processors might remain unchanged from system startup to shutdown. In other systems, Windows might dynamically add active processors while the system is running. Windows never removes active processors from a system.

Thus, the number of active processors in a multiprocessor system can increase between system startup and shutdown, but this number never decreases. Similarly, the number of active groups in the system can increase if a processor is dynamically added to an inactive group while the system is running. The number of active groups never decreases.

In contrast, the value that is returned by KeQueryMaximumGroupCount remains constant during runtime. This value is the maximum number of groups that the system can have.

Call the KeQueryActiveProcessorCountEx routine to determine the number of active logical processors in a particular group.

Windows 7 is the first version of Windows to support processor groups. In Windows 7, only 64-bit versions of Windows support multiple groups; 32-bit versions of Windows support only one group. If a multiprocessor system is running a 64-bit version of Windows and contains no more than 64 logical processors, Windows assigns all processors to group 0. A multiprocessor system that is running a 32-bit version of Windows can contain no more than 32 processors.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryMaximumGroupCount

KeQueryActiveProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessorCount** routine returns the number of currently active processors.

Syntax

```
ULONG KeQueryActiveProcessorCount(
  PKAFFINITY ActiveProcessors
);
```

Parameters

`ActiveProcessors`

A pointer to a **KAFFINITY**-typed variable into which the routine writes a bitmask that represents the set of currently active processors. In a hot-add environment, this mask may change during runtime. This parameter is optional and can be specified as **NULL** if the caller does not require the mask of active processors.

Return Value

KeQueryActiveProcessorCount returns the number of currently active processors.

Remarks

A device driver calls the **KeQueryActiveProcessorCount** routine to retrieve the number of currently active processors. Device drivers that are built for Windows Vista and later versions of the Windows operating system should not use the **KeNumberProcessors** kernel variable for this purpose.

Callers cannot assume that the **KeQueryActiveProcessorCount** routine maps processors to bits in the returned **KAFFINITY** value consecutively, or that the routine will return the same bitmask each time it is called.

Callers must also be aware that the value returned by **KeQueryActiveProcessorCount** can change at runtime on versions of Windows that support hot-add CPU functionality.

If necessary, register for notification of changes to the number of processors by calling the KeRegisterProcessorChangeCallback routine.

To query just the affinity mask without getting the number of active processors, use KeQueryActiveProcessors.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryActiveProcessorCountEx routine, which specifies a processor group, instead of **KeQueryActiveProcessorCount**, which does not. However, the implementation of **KeQueryActiveProcessorCount** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryActiveProcessorCount** returns the number of active logical processors in group 0, and writes an affinity mask to **ActiveProcessors* that specifies the set of active logical processors in group 0.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryActiveProcessors

KeRegisterProcessorChangeCallback

KeQueryActiveProcessorCountEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessorCountEx** routine returns the number of active logical processors in a specified group in a multiprocessor system or in the entire system.

Syntax

```
ULONG KeQueryActiveProcessorCountEx(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

The group number. If a multiprocessor system contains *n* groups, valid group numbers range from 0 to *n*-1. To count all active processors in all groups in the system, set this parameter to ALL_PROCESSOR_GROUPS, which is defined in header files Winnt.h and Ntdef.h.

Return Value

KeQueryActiveProcessorCountEx returns the number of active logical processors in the group. If *GroupNumber* is not a valid group number and is not ALL_PROCESSOR_GROUPS, it returns zero.

Remarks

An active logical processor is a logical processor that Windows has started up and added to a multiprocessor system. The term active processor applies to a processor that is available to perform processing work, regardless of whether the processor is currently performing processing work or is idle. In some systems, the number of active processors might remain unchanged from system startup to shutdown. In other systems, Windows might dynamically add active processors while the system is running. Windows never removes active processors from a system. Thus, the number of active processors in a multiprocessor system can increase between system startup and shutdown, but this number never decreases.

A related routine, KeQueryActiveProcessorCount, returns an active processor count, but this routine, unlike **KeQueryActiveProcessorCountEx**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryActiveProcessorCount** returns the active processor count in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryActiveProcessorCount** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryActiveProcessorCountEx** instead of **KeQueryActiveProcessorCount**.

For a code example that uses **KeQueryActiveProcessorCountEx**, see KeGetProcessorNumberFromIndex.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetProcessorNumberFromIndex

KeQueryActiveProcessorCount

KeQueryActiveProcessors function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryActiveProcessors** routine returns a bitmask of the currently active processors.

Syntax

```
KAFFINITY KeQueryActiveProcessors(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryActiveProcessors returns a **KAFFINITY** value that represents the set of currently active processors.

Remarks

Callers cannot assume that **KeQueryActiveProcessors** maps processors to bits consecutively, or that the routine consistently uses the same mapping each time it is called. The only valid use for the return value is to determine the number of active processors by counting the number of bits that are set.

Callers must also be aware that the value returned by **KeQueryActiveProcessors** can change during runtime on versions of Windows that support hot-add CPU functionality.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryGroupAffinity routine, which specifies a processor group, instead of **KeQueryActiveProcessors**, which does not. However, the implementation of **KeQueryActiveProcessors** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryActiveProcessors** returns an affinity mask that specifies the set of active logical processors in group 0.

In Windows Vista and later versions of Windows, this routine can be called at any IRQL. However, in Windows Server 2003 and earlier versions of Windows, this routine must be called at IRQL <= APC_LEVEL.

The **KeNumberProcessors** kernel variable is obsolete in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of Windows. **KeNumberProcessors** does not appear in WDK headers for WDK releases starting with Windows Vista SP1; however, the variable is still exported from the kernel, so drivers built for earlier platforms will not break

Windows Server 2008 includes support for Dynamic Hardware Partitioning (DHP) in the Windows Datacenter and Enterprise Edition SKUs. As part of DHP, Windows Server 2008 supports hot adding CPUs at runtime. In a hot-add CPU environment, the number of processors may not remain constant during runtime.

Accordingly, in Windows Server 2008, code that can determine the number of processors must use **KeQueryActiveProcessors** instead of direct references to the kernel variable, **KeNumberProcessors**.

Review any code that currently references **KeNumberProcessors** to make sure that it accommodates changes to

CPU count in hot-add CPU environments.

You can use the PNPCPU tool to simulate hot adding a CPU for testing purposes.

C++

```
#if (NTDDI_VERSION >= NTDDI_VISTA)
extern NTSYSAPI volatile CCHAR KeNumberProcessors;
#else
#if (NTDDI_VERSION >= NTDDI_WINXP)
extern NTSYSAPI CCHAR KeNumberProcessors;
#else
extern PCCHAR KeNumberProcessors;
#endif
#endif
```

Starting with Windows XP, **KeNumberProcessors** is an 8-bit integer value that indicates the number of processors in the platform. In earlier versions of Windows, **KeNumberProcessors** is a pointer to an 8-bit integer value that indicates the number of processors in the platform.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlKeApcLte1, HwStorPortProhibitedDDIs

See Also

KeQueryActiveProcessorCount

KeQueryGroupAffinity

KeQueryAuxiliaryCounterFrequency function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryAuxiliaryCounterFrequency** routine returns frequency of the auxiliary counter in units of Hz.

Syntax

```
NTSTATUS KeQueryAuxiliaryCounterFrequency(
  PULONG64 AuxiliaryCounterFrequency
);
```

Parameters

`AuxiliaryCounterFrequency`

A pointer to a variable to which **KeQueryAuxiliaryCounterFrequency** writes the auxiliary counter frequency, in ticks per second. This parameter can be NULL.

Return Value

KeQueryAuxiliaryCounterFrequency can return one of the following:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The query succeeded.
STA TUS _NO T_S UPP ORT ED	Auxiliary counter is not supported.

Remarks

Call this routine to programmatically determine whether auxiliary counter is supported. In that call, if you do not need the frequency of the counter, pass NULL. If not supported, the routine returns STATUS_NOT_SUPPORTED.

Requirements

Minimum supported client	Available starting with Windows 10.

Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Hal.dll
IRQL	Any level

See Also

KeConvertAuxiliaryCounterToPerformanceCounter

KeConvertPerformanceCounterToAuxiliaryCounter

KeQueryDpcWatchdogInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryDpcWatchdogInformation** routine returns the deferred procedure call (DPC) watchdog timer values for the current processor.

Syntax

```
NTSTATUS KeQueryDpcWatchdogInformation(
  PKDPC_WATCHDOG_INFORMATION WatchdogInformation
);
```

Parameters

`WatchdogInformation`

A pointer to a caller-supplied KDPC_WATCHDOG_INFORMATION structure. The routine writes the current DPC watchdog timer values to this structure.

Return Value

KeQueryDpcWatchdogInformation returns an NTSTATUS value. Possible return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	**KeQueryDpcWatchdogInformation** was called while a DPC was running on the current processor.
STATUS_UNSUCCESSFUL	The current processor was not running a DPC when **KeQueryDpcWatchdogInformation** was called.

Remarks

The **KeQueryDpcWatchdogInformation** routine can be called from a DPC routine or at an IRQL=DISPATCH_LEVEL to determine the amount of time that remains before a DPC time-out occurs. The operating system implements a DPC watchdog timer to detect when a single DPC routine runs for too long, or when a series of queued DPC routines runs back-to-back for too long. If DPC time-out errors are enabled, and if either a DPC routine exceeds the time limit for a single routine, or a series of DPC routines exceeds the aggregate time limit, a DPC_WATCHDOG_VIOLATION (0x133) bug check occurs.

KeQueryDpcWatchdogInformation must be called while a DPC is running or invoked at an IRQL = DISPATCH_LEVEL or higher on the current processor. Otherwise, the call fails and returns STATUS_UNSUCCESSFUL.

DPC routines should run only for brief periods, and should delegate as much processing as possible to worker threads. To avoid degrading system responsiveness, a typical DPC routine should run for no more than 100 microseconds each time it is called. For more information, see Guidelines for Writing DPC Routines.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL

See Also

KDPC_WATCHDOG_INFORMATION

KeQueryGroupAffinity function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryGroupAffinity** routine returns an affinity mask that identifies the active logical processors in a specified group in a multiprocessor system.

Syntax

```
KAFFINITY KeQueryGroupAffinity(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

The group number. If a multiprocessor system contains *n* groups, the groups are numbered from 0 to *n*-1. To obtain the number of active groups in the system, call the KeQueryActiveGroupCount routine.

Return Value

KeQueryGroupAffinity returns a **KAFFINITY** value that identifies the active logical processors in the specified group. If the *GroupNumber* parameter value is not a valid group number, the routine returns zero.

Remarks

A related routine, KeQueryActiveProcessors, returns an affinity mask that represents a set of active processors, but this routine, unlike **KeQueryGroupAffinity**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryActiveProcessors** returns an affinity mask for the active processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryActiveProcessors** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryGroupAffinity** instead of **KeQueryActiveProcessors**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level

See Also

KAFFINITY

KeQueryActiveProcessors

KeQueryHighestNodeNumber function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryHighestNodeNumber** routine returns the highest node number in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.

Syntax

```
USHORT KeQueryHighestNodeNumber(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryHighestNodeNumber returns the highest node number.

Remarks

In a non-uniform memory access (NUMA) multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

If a NUMA multiprocessor system contains *n* nodes, the nodes are numbered from 0 to *n*-1, and **KeQueryHighestNodeNumber** returns *n*-1. If a system contains one node, **KeQueryHighestNodeNumber** returns zero.

If a system does not have a NUMA architecture, **KeQueryHighestNodeNumber** returns zero. An example of non-NUMA architecture is a symmetric multiprocessor system (SMP).

When Windows initializes a NUMA multiprocessor system, it assigns logical processors to nodes and counts the number of resulting nodes. Windows defines the capacity of a node as the number of processors that are present when the system starts in addition to any other logical processors that can be added to the node while the system is running. If, during initialization, Windows encounters a NUMA hardware node that contains more logical processors than will fit into a group, Windows splits the node into smaller, logical nodes. Each of these nodes does not exceed the maximum group size. The value that is returned by **KeQueryHighestNodeNumber** indicates the number of logical nodes in the system.

After the system has been initialized, the node count remains fixed while the system continues to run. This count might include memory-only nodes, which are nodes that contain memory but no active logical processors.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

KeQueryInterruptTime function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryInterruptTime** routine returns the current value of the system interrupt time count, with accuracy to within system clock tick.

Syntax

```
ULONGLONG KeQueryInterruptTime(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryInterruptTime returns the current interrupt-time count in 100-nanosecond units. The update to this return value generally occurs at least once per system clock tick.

Remarks

This routine returns the system interrupt time, which is the amount of time since the operating system was last started. The interrupt-time count begins at zero when the operating system starts and is incremented at each clock interrupt by the length of a clock tick. For various reasons, such as hardware differences, the length of a system clock tick can vary between computers. Call the KeQueryTimeIncrement routine to determine the size of a system clock tick.

> **Note** The preceding remarks describe the behavior of **KeQueryInterruptTime** in free (or retail) builds of Windows. In checked (or debug) builds, this behavior is modified to help developers to identify bugs that might not occur until the operating system has been running for many days without being restarted. Specifically, the interrupt-time count and tick count in checked builds are advanced by approximately 49 days. For more information about checked and free builds, see Checked and Free Build Differences.

KeQueryInterruptTime can be used for performance tuning. This routine returns a finer grained measurement than the KeQueryTickCount routine. A call to **KeQueryInterruptTime** has considerably less overhead than a call to the KeQueryPerformanceCounter routine, as well.

Consequently, interrupt time can be used to measure very fine-grained durations while the system is running because operations that set or reset the system time have no effect on the system interrupt time count.

However, power-management state changes do affect the system interrupt time count. Maintenance of the interrupt time count is suspended during system sleep states. When a subsequent wake state transition occurs, the system adds a "bias" value to the interrupt time count to compensate for the estimated duration of such a sleep state. The interrupt time count that is returned by **KeQueryInterruptTime** includes this bias value. To obtain an unbiased interrupt time count, use the KeQueryUnbiasedInterruptTime routine instead of **KeQueryInterruptTime**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

Acquiring high-resolution time stamps

KeQueryInterruptTimePrecise

KeQueryPerformanceCounter

KeQueryTickCount

KeQueryTimeIncrement

KeQueryUnbiasedInterruptTime

KeQueryInterruptTimePrecise function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryInterruptTimePrecise** routine returns the current value of the system interrupt time count, with accuracy to within a microsecond.

Syntax

```
ULONG64 KeQueryInterruptTimePrecise(
  PULONG64 QpcTimeStamp
);
```

Parameters

`QpcTimeStamp`

A pointer to a ULONG64 variable into which the routine writes the performance counter value used to interpolate the return value.

Return Value

The current interrupt-time count in 100-nanosecond units.

Remarks

KeQueryInterruptTimePrecise returns the system's current interrupt time, which is the amount of time since the operating system was last started. **KeQueryInterruptTimePrecise** is similar to the KeQueryInterruptTime routine, but is more precise.

The interrupt time reported by KeQueryInterruptTime is based on the latest tick of the system clock timer. The clock timer is the hardware timer that periodically generates interrupts for the system clock. The uniform period between clock timer interrupts is referred to as a system clock tick, and is typically in the range of 500 microseconds to 15.625 milliseconds, depending on the hardware platform. The interrupt time value retrieved by **KeQueryInterruptTime** is accurate within a system clock tick.

To provide an interrupt time value that is more precise than that of KeQueryInterruptTime, **KeQueryInterruptTimePrecise** uses the system performance counter to measure the time elapsed since the last clock timer interrupt, and adds this time to the interrupt time associated with the latest clock time. The interrupt time reported by **KeQueryInterruptTimePrecise** is accurate to within a microsecond.

On some hardware platforms, a **KeQueryInterruptTimePrecise** call might be slower than a KeQueryInterruptTime call. The reason is that **KeQueryInterruptTimePrecise** reads the performance counter, which can introduce an additional delay. For more information, see KeQueryPerformanceCounter.

Call the KeQueryTimeIncrement routine to determine the size of a system clock tick.

> **Note** The preceding remarks describe the behavior of **KeQueryInterruptTime** in free (or retail) builds of Windows. In checked (or debug) builds, this behavior is modified to help developers to identify bugs that might not occur until the operating system has been running for many days without being restarted. Specifically, the interrupt-time count and tick count in checked builds are advanced by approximately 49 days. For more information about checked and free builds, see Checked and Free Build Differences.

Precise interrupt time can be used to measure very fine-grained durations while the system is running because operations that set or reset the system time have no effect on the system interrupt time count.

However, power-management state changes do affect the system interrupt time count. Maintenance of the interrupt time count is suspended during system sleep states. When a subsequent wake state transition occurs, the system adds a "bias" value to the interrupt time count to compensate for the estimated duration of such a sleep state. The interrupt time count that is returned by **KeQueryInterruptTimePrecise** includes this bias value. To obtain an unbiased interrupt time count, use the KeQueryUnbiasedInterruptTime. A precise version of the unbiased interrupt time count is not currently available.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

Acquiring high-resolution time stamps

KeQueryInterruptTime

KeQueryTimeIncrement

KeQueryUnbiasedInterruptTime

KeQueryLogicalProcessorRelationship function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryLogicalProcessorRelationship** routine gets information about the relationships of one or more processors to the other processors in a multiprocessor system.

Syntax

```
NTSTATUS KeQueryLogicalProcessorRelationship(
  PPROCESSOR_NUMBER                       ProcessorNumber,
  LOGICAL_PROCESSOR_RELATIONSHIP          RelationshipType,
  PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX Information,
  PULONG                                  Length
);
```

Parameters

`ProcessorNumber`

A pointer to a PROCESSOR_NUMBER structure that identifies the logical processor for which the caller requests relationship information. To request information about all logical processors in the system, set this parameter to **NULL**.

`RelationshipType`

Specifies the type of relationship information that is requested by the caller. Set this parameter to one of the following LOGICAL_PROCESSOR_RELATIONSHIP enumeration values:

- **RelationProcessorCore**
- **RelationNumaNode**
- **RelationCache**
- **RelationProcessorPackage**
- **RelationGroup**
- **RelationAll**

`Information`

A pointer to a caller-allocated buffer into which the routine writes an array of one or more SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structures that contain the information requested by the caller. If the function fails, the contents of this buffer are undefined. Set *Information* = **NULL** to obtain the required buffer length before you allocate the buffer. For more information, see the following Remarks section.

`Length`

A pointer to a location that contains the size, in bytes, of the buffer that is pointed to by *Information*. On entry, * *Length* contains the size of the caller-allocated buffer that is pointed to by *Information*. During the call, the routine overwrites the value that is pointed to by *Length* with the buffer size that is required to contain the requested relationship information.

Return Value

KeQueryLogicalProcessorRelationship returns STATUS_SUCCESS if the call is successful. Possible error return

values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ProcessorNumber* parameter points to a PROCESSOR_NUMBER structure that contains invalid information.
STATUS_INFO_LENGTH_MISMATCH	The caller-allocated buffer that is pointed to by the *Information* parameter is not large enough to contain the requested relationship information.

Remarks

To determine the buffer size to allocate, initially call **KeQueryLogicalProcessorRelationship** with *Information* = **NULL** and **Length* = 0. In response, the routine writes the required buffer size to **Length* and returns STATUS_INFO_LENGTH_MISMATCH. Next, allocate a buffer of the required size and call **KeQueryLogicalProcessorRelationship** a second time. In this second call, set *Information* to the buffer address and **Length* to the buffer size. If the second call succeeds, the routine writes the requested relationship information to the buffer and returns STATUS_SUCCESS.

Examples

The following code example gets processor relationship information for all the logical processors in a multiprocessor system:

```
//
// Get required buffer size.
//
NTSTATUS Status;
PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX Info = NULL;
ULONG BufferSize = 0;
Status = KeQueryLogicalProcessorRelationship(NULL, RelationAll, NULL, &BufferSize);

NT_ASSERT(Status == STATUS_INFO_LENGTH_MISMATCH && BufferSize > 0);

//
// Allocate buffer (assume IRQL <= APC_LEVEL).
//
Info = ExAllocatePoolWithTag(PagedPool, BufferSize, ' gaT');
if (Info == NULL)
{
    Status = STATUS_INSUFFICIENT_RESOURCES;
}

//
// Get processor relationship information.
//
if (NT_SUCCESS(Status))
{
    Status = KeQueryLogicalProcessorRelationship(NULL, RelationAll, Info, &BufferSize);
}
```

The NT_ASSERT macro is defined in the Wdm.h header file. The NT_SUCCESS macro is defined in the Ntdef.h header file.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

LOGICAL_PROCESSOR_RELATIONSHIP

PROCESSOR_NUMBER

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

KeQueryMaximumGroupCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumGroupCount** routine returns the maximum number of groups in a multiprocessor system.

Syntax

```
USHORT KeQueryMaximumGroupCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryMaximumGroupCount returns the maximum number of groups.

Remarks

The value that is returned by **KeQueryMaximumGroupCount** remains constant during runtime. This value depends on the hardware configuration of the multiprocessor system, but it can never exceed a fixed limit that is set by the Windows operating system.

In Windows 7, the maximum number of groups in a multiprocessor system is four, but this value might change in future versions of Windows. The safest way to determine the maximum number of groups in Windows 7 or a later versions of the Windows operating system is to call **KeQueryMaximumGroupCount**. Kernel-mode drivers that call **KeQueryMaximumGroupCount** will not require code changes if the formula that is used to calculate the maximum number of groups changes in a future version of Windows.

To obtain the number of active groups in a multiprocessor system, call the KeQueryActiveGroupCount routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveGroupCount

KeQueryMaximumProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumProcessorCount** routine returns the maximum number of processors.

Syntax

```
ULONG KeQueryMaximumProcessorCount(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryMaximumProcessorCount returns the maximum number of processors as a ULONG value.

Remarks

The value returned by the **KeQueryMaximumProcessorCount** routine does not change at runtime.

If your code uses an array of buffers, one buffer for each processor, you must decide whether to have a statically sized array based on **KeQueryMaximumProcessorCount** or a dynamically sized array based on KeQueryActiveProcessorCount.

To optimize based on the number of processors, you need a resizable structure for Windows Server 2008. In this case, use **KeQueryActiveProcessorCount**.

If you are not optimizing and if the data structures that result from using maximum processor count are relatively small, a resizable structure is not necessary. In this case, use **KeQueryMaximumProcessorCount** to determine size for a static array.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeQueryMaximumProcessorCountEx routine, which specifies a processor group, instead of **KeQueryMaximumProcessorCount**, which does not. However, the implementation of **KeQueryMaximumProcessorCount** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeQueryMaximumProcessorCount** returns the maximum number of logical processors that can be in group 0.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCount

KeQueryMaximumProcessorCountEx

KeQueryMaximumProcessorCountEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryMaximumProcessorCountEx** routine returns the maximum number of logical processors in a specified group in a multiprocessor system.

Syntax

```
ULONG KeQueryMaximumProcessorCountEx(
  USHORT GroupNumber
);
```

Parameters

`GroupNumber`

Specifies a group number. If a multiprocessor system contains *n* groups, the groups are numbered from 0 to *n*-1. To obtain the maximum number of processors in the system, set this parameter to ALL_PROCESSOR_GROUPS, which is defined in the Winnt.h and Ntdef.h header files.

Return Value

KeQueryMaximumProcessorCountEx returns the maximum number of logical processors in the specified group or in the system, as indicated by the value of the *GroupNumber* parameter.

Remarks

The value that is returned by **KeQueryMaximumProcessorCountEx** remains constant during runtime. This value depends on the hardware configuration of the multiprocessor system, but it can never exceed a fixed limit that is set by the Windows operating system. This limit is 32 logical processors for 32-bit versions of Windows and 64 logical processors for 64-bit versions of Windows.

In contrast, the value that is returned by the KeQueryActiveProcessorCountEx routine can change during runtime as processors are dynamically added to the system.

A related routine, KeQueryMaximumProcessorCount, returns a maximum processor count, but this routine, unlike **KeQueryMaximumProcessorCountEx**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeQueryMaximumProcessorCount** returns the maximum number of logical processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeQueryMaximumProcessorCount** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeQueryMaximumProcessorCountEx** instead of **KeQueryMaximumProcessorCount**.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryActiveProcessorCountEx

KeQueryMaximumProcessorCount

KeQueryNodeActiveAffinity function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryNodeActiveAffinity** routine gets the current processor affinity of a specified node in a multiprocessor system that has a non-uniform memory access (NUMA) architecture.

Syntax

```
void KeQueryNodeActiveAffinity(
  USHORT          NodeNumber,
  PGROUP_AFFINITY Affinity,
  PUSHORT         Count
);
```

Parameters

`NodeNumber`

The node number. If a multiprocessor system contains *n* nodes, the nodes are numbered from 0 to *n*-1. To obtain the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

`Affinity`

A pointer to a caller-allocated buffer into which the routine writes a GROUP_AFFINITY structure. This structure contains the group number of the group that contains the node that is identified by *NodeNumber*, and an affinity mask that indicates which logical processors in the node are active. You can set this parameter to **NULL** if you do not need this information.

`Count`

A pointer to a location into which the routine writes the number of active processors that are represented in the node affinity mask that is pointed to by *Affinity*. You can set this parameter to **NULL** if you do not need this information.

Return Value

None

Remarks

The caller-allocated buffer that is pointed to by the *Affinity* parameter must be large enough to contain a **GROUP_AFFINITY** structure. The **Mask** member of this structure contains an affinity mask that indicates which processors are active. If a processor is active, the corresponding bit in the mask is one. All other bits are zero.

In a NUMA multiprocessor architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

The number of processors in a node cannot exceed the number of bits in the affinity mask in the structure that is pointed to by *Affinity*. The affinity mask also determines the maximum number of processors in a group.

If, during system initialization, Windows encounters a NUMA hardware node that contains more logical processors

than will fit into a group, Windows splits the node into smaller, logical nodes. Each of these nodes does not exceed the maximum group size. The *NodeNumber* parameter identifies a logical node. To obtain the maximum number of processors per group, call the KeQueryMaximumProcessorCountEx routine.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

GROUP_AFFINITY

KeQueryHighestNodeNumber

KeQueryMaximumProcessorCountEx

KeQueryNodeMaximumProcessorCount function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryNodeMaximumProcessorCount** routine returns the maximum number of logical processors that a specified node in a non-uniform memory access (NUMA) multiprocessor system can contain.

Syntax

```
USHORT KeQueryNodeMaximumProcessorCount(
  USHORT NodeNumber
);
```

Parameters

`NodeNumber`

The node number. If a NUMA multiprocessor system contains *n* nodes, valid node numbers are in the range 0 to *n*-1. To get the highest node number (*n*-1) in the system, call the KeQueryHighestNodeNumber routine.

Return Value

KeQueryNodeMaximumProcessorCount returns the maximum number of logical processors.

Remarks

In a multiprocessor system with a NUMA architecture, a node is a collection of processors that share fast access to a region of memory. Memory access is non-uniform because a processor can access the memory in its node faster than it can access the memory in other nodes.

The count returned by this routine includes any logical processors that can be dynamically added to the node while the multiprocessor system is running.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryHighestNodeNumber

KeQueryPerformanceCounter function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryPerformanceCounter** routine retrieves the current value and frequency of the performance counter.

Use **KeQueryPerformanceCounter** to acquire high resolution (<1us) time stamps for time interval measurements.

Syntax

```
NTHALAPI LARGE_INTEGER KeQueryPerformanceCounter(
  PLARGE_INTEGER PerformanceFrequency
);
```

Parameters

`PerformanceFrequency`

A pointer to a variable to which **KeQueryPerformanceCounter** writes the performance counter frequency, in ticks per second. This parameter is optional and can be NULL if the caller does not need the counter frequency value.

Return Value

KeQueryPerformanceCounter returns the performance counter value in units of ticks.

Remarks

KeQueryPerformanceCounter returns a 64-bit integer that represents the current value of a high-resolution monotonically nondecreasing counter.

To obtain the frequency of the performance counter, specify a non-**NULL** pointer value for the *PerformanceFrequency* parameter. The frequency of the performance counter is fixed at system boot and is consistent across all processors. Therefore, a driver can cache the frequency of the performance counter during initialization.

For more info about this function and its usage, see Acquiring high-resolution time stamps.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
DLL	Hal.dll

IRQL	Any level

See Also

KeQueryInterruptTime

KeQuerySystemTime

KeQueryTickCount

KeQueryTimeIncrement

QueryPerformanceCounter

QueryPerformanceFrequency

KeQueryPriorityThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryPriorityThread** routine returns the current priority of a particular thread.

Syntax

```
KPRIORITY KeQueryPriorityThread(
  PKTHREAD Thread
);
```

Parameters

`Thread`

Pointer to a dispatcher object of type KTHREAD.

Return Value

KeQueryPriorityThread returns the current priority of the specified thread.

Remarks

Thread priorities range from 0 to 31, where 0 is the lowest priority and 31 is the highest.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeGetCurrentThread

KeSetBasePriorityThread

KeSetPriorityThread

PsGetCurrentThread

KeQueryRuntimeThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryRuntimeThread** routine reports the accumulated kernel-mode and user-mode run time of a thread, in clock ticks.

Syntax

```
ULONG KeQueryRuntimeThread(
  PKTHREAD Thread,
  PULONG   UserTime
);
```

Parameters

`Thread`

Pointer to a dispatcher object of type KTHREAD.

`UserTime`

Pointer to the memory location where **KeQueryRuntimeThread** returns the accumulated user-mode run time of the current thread, in clock ticks.

Return Value

KeQueryRuntimeThread returns the accumulated kernel-mode run time of the current thread, in clock ticks.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeQueryTimeIncrement

KeQuerySystemTime function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQuerySystemTime** routine obtains the current system time.

Syntax

```
void KeQuerySystemTime(
  PLARGE_INTEGER CurrentTime
);
```

Parameters

`CurrentTime`

Return Value

None

Remarks

System time is a count of 100-nanosecond intervals since January 1, 1601. System time is typically updated approximately every ten milliseconds. This value is computed for the GMT time zone. To adjust this value for the local time zone use ExSystemTimeToLocalTime.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ExSystemTimeToLocalTime

KeQueryPerformanceCounter

KeQueryTickCount

KeQueryTimeIncrement

KeQuerySystemTimePrecise function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQuerySystemTimePrecise** routine retrieves the current system time, and is more precise than the KeQuerySystemTime routine.

Syntax

```
void KeQuerySystemTimePrecise(
  PLARGE_INTEGER CurrentTime
);
```

Parameters

`CurrentTime`

A pointer to a LARGE_INTEGER variable into which the routine writes the current system time.

Return Value

None

Remarks

System time is a count of 100-nanosecond intervals since January 1, 1601. This value is computed for the GMT time zone. To adjust this value for the local time zone, call the ExSystemTimeToLocalTime routine.

KeQuerySystemTimePrecise is similar to the **KeQuerySystemTime** routine, but is more precise. The system time reported by **KeQuerySystemTime** is based on the latest tick of the system clock timer. The clock timer is the hardware timer that periodically generates interrupts for the system clock. The uniform period between clock timer interrupts is referred to as a system clock tick, and is typically in the range of 500 microseconds to 15.625 milliseconds, depending on the hardware platform. The system time value retrieved by **KeQuerySystemTime** is accurate within a system clock tick.

To provide a system time value that is more accurate than that of **KeQuerySystemTime**, **KeQuerySystemTimePrecise** uses the system performance counter to measure the time elapsed since the last clock timer interrupt, and adds this time to the system clock time that it calculates from the clock timer count. The system time reported by **KeQuerySystemTimePrecise** is accurate to within a microsecond.

On some hardware platforms, a **KeQuerySystemTimePrecise** call might be slower than a **KeQuerySystemTime** call. The reason is that **KeQuerySystemTimePrecise** reads the performance counter, which can introduce an additional delay. For more information, see KeQueryPerformanceCounter.

Call the KeQueryTimeIncrement routine to determine the duration of a system clock tick.

Requirements

Minimum supported client	Available starting with Windows 8.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ExSystemTimeToLocalTime

KeQueryPerformanceCounter

KeQuerySystemTime

KeQueryTickCount

KeQueryTimeIncrement

KeQueryTickCount macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryTickCount** routine maintains a count of the interval timer interrupts that have occurred since the system was booted.

Syntax

```
void KeQueryTickCount(
   CurrentCount
);
```

Parameters

`CurrentCount`

Pointer to the LARGE_INTEGER tick count value on return from **KeQueryTickCount**.

Return Value

None

Remarks

The *TickCount* value increases by one at each interval timer interrupt while the system is running.

The preferred method of determining elapsed time is by using *TickCount* for relative timing and time stamps.

To determine the absolute elapsed time multiply the returned *TickCount* by the KeQueryTimeIncrement return value using compiler support for 64-bit integer operations.

You should not make any assumptions about the length of a tick, because it might vary depending on hardware and other considerations.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryInterruptTime

KeQueryPerformanceCounter

KeQueryTimeIncrement

KeQueryTimeIncrement function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryTimeIncrement** routine returns the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.

Syntax

```
ULONG KeQueryTimeIncrement(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryTimeIncrement returns a ULONG value indicating the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.

Remarks

At startup time, the operating system determines the time increment to use for the system time. This time increment remains constant until the computer restarts. During this time, calls to **KeQueryTimeIncrement** always return the same time increment value. The time increment does not change while the computer is running, and it does not change as the result of a suspend-resume cycle.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryPerformanceCounter

KeQuerySystemTime

KeQueryTickCount

KeQueryTotalCycleTimeThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryTotalCycleTimeThread** routine returns the accumulated cycle time for the specified thread.

Syntax

```
ULONG64 KeQueryTotalCycleTimeThread(
  PKTHREAD Thread,
  PULONG64 CycleTimeStamp
);
```

Parameters

`Thread`

A pointer to a dispatcher object of type KTHREAD.

`CycleTimeStamp`

A pointer to the cycle counter value at the time of the query.

Return Value

The accumulated cycle time for the thread.

Requirements

Minimum supported client	Available in Windows 8 and later versions of Windows.
Target Platform	Universal
Header	wdm.h

KeQueryUnbiasedInterruptTime function

1/25/2019 • 2 minutes to read • Edit Online

The **KeQueryUnbiasedInterruptTime** routine returns the current value of the system interrupt time count.

Syntax

```
ULONGLONG KeQueryUnbiasedInterruptTime(

);
```

Parameters

This function has no parameters.

Return Value

KeQueryUnbiasedInterruptTime returns the current interrupt time count in 100-nanosecond units. The count begins at zero when the computer starts. Updates to this count are suspended when the computer enters a sleep state and are resumed when the computer awakens.

Remarks

Kernel-mode drivers can call this routine to measure relatively fine-grained durations.

This routine returns the system interrupt time, which is the amount of time since the operating system was last started. The interrupt-time count begins at zero when the operating system starts and is incremented at each clock interrupt by the length of a clock tick. For various reasons, such as hardware differences, the length of a system clock tick can vary between computers. Call the KeQueryTimeIncrement routine to determine the size of a system clock tick.

> **Note** The preceding remarks describe the behavior of **KeQueryUnbiasedInterruptTime** in free (or retail) builds of Windows. In checked (or debug) builds, this behavior is modified to help developers to identify bugs that might not occur until the operating system has been running for many days without being restarted. Specifically, the interrupt-time count and tick count in checked builds are advanced by approximately 49 days. For more information about checked and free builds, see Checked and Free Build Differences.

The **KeQueryUnbiasedInterruptTime** and KeQueryInterruptTime routines are similar, but they differ in the way in which they account for periods in which the operating system suspends updates of the interrupt time count. For example, these updates are suspended while the computer is in a system sleep state. To compensate for periods in which updates of the count are suspended, **KeQueryInterruptTime** returns a biased count value that it calculates by adding a bias count to the raw interrupt time count. The bias count is the estimated sum of all of the updates to the interrupt time count that the operating system missed while counting was suspended. In contrast, **KeQueryUnbiasedInterruptTime** returns the raw, unbiased count.

Unlike the KeQuerySystemTime routine, **KeQueryUnbiasedInterruptTime** returns a count value that is not affected by operations that set or reset the system time. In addition, the system time that is reported by **KeQuerySystemTime** is typically updated approximately every ten milliseconds. In contrast, the count that is returned by **KeQueryUnbiasedInterruptTime** is updated at least once per system clock tick.

In Windows 2000 and later versions of the Windows operating system, the KeQueryPerformanceCounter routine provides the finest grained running count that is available from the operating system.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeQueryInterruptTime

KeQueryInterruptTimePrecise

KeQueryPerformanceCounter

KeQuerySystemTime

KeQueryTimeIncrement

KeRaiseIrql macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeRaiseIrql** routine raises the hardware priority to the specified IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor. For information about IRQLs, see Managing Hardware Priorities.

Syntax

```
void KeRaiseIrql(
   a,
   b
);
```

Parameters

`a`

Specifies the new KIRQL value to which the hardware priority is to be raised.

`b`

Pointer to the storage for the original (unraised) KIRQL value to be used in a subsequent call to KeLowerIrql.

Return Value

None

Remarks

If the new IRQL is less than the current IRQL, a bug check occurs. Otherwise, the current IRQL is set to the specified value.

Callers of this routine can be running at any IRQL. Any caller should restore the original IRQL with KeLowerIrql as soon as possible.

A call to **KeLowerIrql** is valid if it specifies *NewIrql* <= *CurrentIrql*. A call to **KeRaiseIrql** is valid if the caller specifies *NewIrql* >= *CurrentIrql*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)
DDI compliance rules	IrqlKeRaiseLower, IrqlKeRaiseLower2, HwStorPortProhibitedDDIs

See Also

KeGetCurrentIrql

KeLowerIrql

KeRaiseIrqlToDpcLevel function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRaiseIrqlToDpcLevel** routine raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

Syntax

```
_DECL_HAL_KE_IMPORT KIRQL KeRaiseIrqlToDpcLevel(

);
```

Parameters

This function has no parameters.

Return Value

KeRaiseIrqlToDpcLevel returns the IRQL at which the call occurred.

Remarks

Any caller of **KeRaiseIrqlToDpcLevel** should save the returned IRQL value. Every such caller must restore the original IRQL as quickly as possible by passing this returned IRQL in a subsequent call to KeLowerIrql.

Callers of **KeRaiseIrqlToDpcLevel** must be running at IRQL <= DISPATCH_LEVEL. Otherwise, a call to this routine causes a bug check.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	Hal.lib
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeGetCurrentIrql

KeLowerIrql

KeRaiseIrql

KeReadStateEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReadStateEvent** routine returns the current state, signaled or not-signaled, of an event object.

Syntax

```
LONG KeReadStateEvent(
  PRKEVENT Event
);
```

Parameters

`Event`

A pointer to an initialized event object for which the caller provides the storage.

Return Value

If the event object is currently set to a signaled state, a nonzero value is returned. Otherwise, zero is returned.

Remarks

This routine provides an efficient way to poll the signal state of an event. **KeReadStateEvent** reads the state of the event without synchronizing its access to the event. Do not assume that accesses of an event state by **KeReadStateEvent** are mutually exclusive of accesses by routines, such as KeSetEvent and KeWaitForSingleObject, that do synchronize their access to the event state.

It is also possible to read the state of an event from a driver's interrupt service routine at DIRQL, if the following conditions are met: the driver's event object is resident (probably in its device extension), and any other function that accesses the event synchronizes its access with the ISR.

For more information about event objects, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeClearEvent

KeInitializeEvent

KeResetEvent

KeSetEvent

KeWaitForSingleObject

KeReadStateMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReadStateMutex** routine returns the current state, signaled or not-signaled, of the specified mutex object.

Syntax

```
LONG KeReadStateMutex(
  PRKMUTEX Mutex
);
```

Parameters

`Mutex`

Pointer to an initialized mutex object for which the caller provides the storage.

Return Value

If the return value is 1, the state of the mutex object is signaled.

Remarks

This routine provides an efficient way to poll the signal state of a mutex. **KeReadStateMutex** reads the state of the mutex without synchronizing its access to the mutex. Do not assume that accesses of a mutex state by **KeReadStateMutex** are mutually exclusive of accesses by routines, such as KeReleaseMutex and KeWaitForSingleObject, that do synchronize their access to the mutex state.

For more information about mutex objects, see Mutex Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ExInitializeFastMutex

KeInitializeMutex

KeReleaseMutex

KeWaitForSingleObject

KeReadStateSemaphore function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReadStateSemaphore** routine returns the current state, signaled or not-signaled, of the specified semaphore object.

Syntax

```
LONG KeReadStateSemaphore(
  PRKSEMAPHORE Semaphore
);
```

Parameters

`Semaphore`

Pointer to an initialized semaphore object for which the caller provides the storage.

Return Value

If the return value is zero, the semaphore object is set to a not-signaled state.

Remarks

This routine provides an efficient way to poll the signal state of a semaphore. **KeReadStateSemaphore** reads the state of the semaphore without synchronizing its access to the semaphore. Do not assume that accesses of a semaphore state by **KeReadStateSemaphore** are mutually exclusive of accesses by routines, such as KeReleaseSemaphore and KeWaitForSingleObject, that do synchronize their access to the semaphore state.

For more information about semaphore objects, see Semaphore Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeInitializeSemaphore

KeReleaseSemaphore

KeWaitForSingleObject

KeReadStateTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReadStateTimer** routine reads the current state of a timer object.

Syntax

```
BOOLEAN KeReadStateTimer(
  PKTIMER Timer
);
```

Parameters

`Timer`

A pointer to an initialized timer object, for which the caller provides the storage.

Return Value

KeReadStateTimer returns **TRUE** if the current state of the timer object is signaled; otherwise, it returns **FALSE**.

Remarks

This routine provides an efficient way to poll the signal state of a timer. **KeReadStateTimer** reads the state of the timer without synchronizing its access to the timer. Do not assume that accesses of a timer state by **KeReadStateTimer** are mutually exclusive of accesses by routines, such as KeSetTimer and KeWaitForSingleObject, that do synchronize their access to the timer state.

For more information about timer objects, see Timer Objects and DPCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KTIMER

KeCancelTimer

KeInitializeTimer

KeSetTimer

KeWaitForSingleObject

KeRegisterBoundCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRegisterBoundCallback** routine registers a routine to be called whenever a user-mode bound exception occurs.

Syntax

```
PVOID KeRegisterBoundCallback(
  PBOUND_CALLBACK CallbackRoutine
);
```

Parameters

`CallbackRoutine`

Pointer to a BoundCallback function.

Return Value

On success, **KeRegisterBoundCallback** returns an opaque pointer that the caller passes to KeDeregisterBoundCallback to deregister the callback. The routine returns **NULL** if it is unable to register the callback.

Remarks

The **KeRegisterBoundCallback** routine provides the opportunity for a kernel-mode driver to intercept and handle user-mode bound exceptions. When a bounds exception for a user-mode thread occurs, the system calls the registered BoundCallback function to manage the bounds trap. The return value of the *BoundCallback* function indicates the action that the system should then perform, such as propagating the bounds exception or terminating the user-mode process.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

See Also

KeDeregisterBoundCallback

KeRegisterBugCheckCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRegisterBugCheckCallback** routine registers a BugCheckCallback routine, which executes when the operating system issues a bug check.

Syntax

```
BOOLEAN KeRegisterBugCheckCallback(
  PKBUGCHECK_CALLBACK_RECORD  CallbackRecord,
  PKBUGCHECK_CALLBACK_ROUTINE CallbackRoutine,
  PVOID                       Buffer,
  ULONG                       Length,
  PUCHAR                      Component
);
```

Parameters

`CallbackRecord`

Pointer to a callback record that was previously initialized by KeInitializeCallbackRecord, for which the caller provides nonpaged storage.

`CallbackRoutine`

Pointer to the driver-supplied BugCheckCallback routine.

`Buffer`

Pointer to a caller-supplied buffer, which must be in resident memory, such as nonpaged pool.

`Length`

Specifies the size in bytes of the caller-allocated buffer.

`Component`

Pointer to a null-terminated ANSI string that identifies the caller. This string usually contains the name of the device driver, or possibly of its device.

Return Value

KeRegisterBugCheckCallback returns **TRUE** if the caller-supplied routine is successfully added to the set of registered bug-check callbacks; otherwise, it returns **FALSE**.

Remarks

The **KeRegisterBugCheckCallback** routine registers a BugCheckCallback routine, the simplest kind of bug-check callback routine. To register other kinds of bug-check callbacks, such as BugCheckDumpIoCallback and BugCheckAddPagesCallback routines, use the KeRegisterBugCheckReasonCallback routine instead.

The *BugCheckCallback* routine is executed when the system issues a bug check. A driver can use the routine to reset the device to a known state. For more information, see *BugCheckCallback*.

Drivers can use the **KeDeregisterBugCheckCallback** routine to remove the callback. Any driver that can be unloaded must remove all of its callbacks in its Unload routine.

The *Component* string is used to identify the driver during crash-dump debugging. To display the crash dump data corresponding to any specified *Component* string, you can use the **!bugdump** debugger extension. It is also possible to debug the bug check callback routine itself. For information about debuggers and debugger extensions, see Windows Debugging.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

BugCheckAddPagesCallback

BugCheckCallback

BugCheckDumpIoCallback

KeDeregisterBugCheckCallback

KeInitializeCallbackRecord

KeRegisterBugCheckReasonCallback

KeRegisterBugCheckReasonCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRegisterBugCheckReasonCallback** routine registers a BugCheckDumpIoCallback, BugCheckSecondaryDumpDataCallback, or BugCheckAddPagesCallback routine, which executes when the operating system issues a bug check.

Syntax

```
BOOLEAN KeRegisterBugCheckReasonCallback(
  PKBUGCHECK_REASON_CALLBACK_RECORD  CallbackRecord,
  PKBUGCHECK_REASON_CALLBACK_ROUTINE CallbackRoutine,
  KBUGCHECK_CALLBACK_REASON          Reason,
  PUCHAR                             Component
);
```

Parameters

`CallbackRecord`

A pointer to a KBUGCHECK_REASON_CALLBACK_RECORD structure that was initialized by a call to the KeInitializeCallbackRecord routine.

`CallbackRoutine`

A pointer to the callback routine to register. This parameter points to one of the following types of driver-implemented routine:

BugCheckDumpIoCallback

BugCheckSecondaryDumpDataCallback

BugCheckAddPagesCallback

`Reason`

Specifies the type of callback routine that *CallbackRoutine* points to. Set *Reason* to one of the KBUGCHECK_CALLBACK_REASON enumeration values in the following table.

VALUE	CALLBACK ROUTINE
KbCallbackDumpIo	*BugCheckDumpIoCallback*
KbCallbackSecondaryDumpData	*BugCheckSecondaryDumpDataCallback*
KbCallbackAddPages	*BugCheckAddPagesCallback*

`Component`

A pointer to a null-terminated ANSI string that identifies the caller. For example, you can select a string that describes the device driver, or that contains the device name. You can use the !bugdump debugger extension to display the crash dump data that is associated with this string.

Return Value

KeRegisterBugCheckReasonCallback returns **TRUE** if the callback routine is successfully registered; otherwise, it returns **FALSE**.

Remarks

Drivers can use **KeRegisterBugCheckReasonCallback** to register routines that execute during a system bug check.

BugCheckDumpIoCallback routines are called each time data is written to the crash dump file. Drivers for devices that monitor the system state can register a *BugCheckDumpIoCallback* routine to copy the crash dump data to the monitoring device.

BugCheckSecondaryDumpDataCallback routines are called to poll drivers for any device-specific information that should be added to the secondary data area of a crash dump file.

BugCheckAddPagesCallback routines are called to enable drivers to add pages of driver-specific data to the primary data area of a crash dump file. *BugCheckAddPagesCallback* routines are supported in Windows Server 2008 and later versions of Windows.

Drivers can use the KeDeregisterBugCheckReasonCallback routine to remove the *BugCheckXxxCallback* registration. Any driver that can be unloaded must remove the registrations of all of its callbacks in its Unload routine.

To display secondary dump data, you can use the .enumtag command or the IDebugDataSpaces3::ReadTagged method in a debugger extension. Another option is to debug the bug check callback routine itself. For more information about debuggers and debugger extensions, see Windows Debugging.

Requirements

Minimum supported client	Available in Windows XP Service Pack 1 (SP1), Windows Server 2003, and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

BugCheckAddPagesCallback

BugCheckCallback

BugCheckDumpIoCallback

BugCheckSecondaryDumpDataCallback

KBUGCHECK_CALLBACK_REASON

KBUGCHECK_REASON_CALLBACK_RECORD

KeDeregisterBugCheckReasonCallback

KeInitializeCallbackRecord

KeRegisterBugCheckCallback

KeRegisterNmiCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRegisterNmiCallback** routine registers a routine to be called whenever a nonmaskable interrupt (NMI) occurs.

Syntax

```
PVOID KeRegisterNmiCallback(
  PNMI_CALLBACK CallbackRoutine,
  PVOID         Context
);
```

Parameters

`CallbackRoutine`

Pointer to a function of the form:

```
BOOLEAN
 XxxNmiCallback(
 IN PVOID Context,
  IN BOOLEAN Handled
 );
```

`Context`

Specifies the value to be passed as the *Context* parameter of *XxxNmiCallback* when it is called.

Return Value

On success, **KeRegisterNmiCallback** returns an opaque pointer that the caller passes to KeDeregisterNmiCallback to deregister the callback. The routine returns **NULL** if it is unable to register the callback.

Remarks

When a nonmaskable interrupt occurs, the system calls each registered callback in reverse order from the order in which they were registered. For the first callback, the system passes **FALSE** as the *Handled* parameter. For each subsequent callback, if any previous callback returned **TRUE**, the system passes **TRUE** as the *Handled* parameter, otherwise it passes **FALSE**. If any callback returns a value of **TRUE**, the system considers the interrupt to have been handled. Otherwise, the system calls the HAL's default handler for the interrupt, which normally causes the system to bug check.

The callback routine must be able to be run at IRQL = HIGH_LEVEL.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (see Remarks section)"
DDI compliance rules	IrqlKeApcLte2, HwStorPortProhibitedDDIs

See Also

KeDeregisterNmiCallback

KeRegisterProcessorChangeCallback function

1/25/2019 • 6 minutes to read • Edit Online

The **KeRegisterProcessorChangeCallback** routine registers a callback function with the operating system so that the operating system will notify the driver when a new processor is added to the hardware partition.

Syntax

```
PVOID KeRegisterProcessorChangeCallback(
  PPROCESSOR_CALLBACK_FUNCTION CallbackFunction,
  PVOID                        CallbackContext,
  ULONG                        Flags
);
```

Parameters

`CallbackFunction`

A pointer to a driver-supplied processor change callback function that is to be called by the operating system whenever a new processor is added to the hardware partition. A processor change callback function is defined as follows:

```
VOID
  ProcessorCallback(
    __in PVOID CallbackContext,
    __in PKE_PROCESSOR_CHANGE_NOTIFY_CONTEXT ChangeContext,
    __inout PNTSTATUS OperationStatus
    );
```

CallbackContext

The context that was supplied in the *CallbackContext* parameter to the **KeRegisterProcessorChangeCallback** routine when the callback function was registered with the operating system.

ChangeContext

A pointer to a KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that describes the processor change notification event.

OperationStatus

A pointer to a variable that contains an NTSTATUS code. A device driver must not change the value of this variable except if an error occurs during the processing of the callback function when the **State** member of the KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains **KeProcessorAddStartNotify**.

The processor change callback function is called at IRQL = PASSIVE_LEVEL.

`CallbackContext`

A driver-supplied context that is passed to the callback function. This parameter can be **NULL**.

`Flags`

Optional flags that modify the behavior of the **KeRegisterProcessorChangeCallback** routine. The following is one possible flag:

KE_PROCESSOR_CHANGE_ADD_EXISTING

If this flag is set, the registered callback function is immediately called for each active processor that currently exists in the hardware partition, in addition to being called whenever a new processor is added to the hardware partition. If this flag is not set, the registered callback function is only called whenever a new processor is added to the system.

Return Value

KeRegisterProcessorChangeCallback returns a non-**NULL** callback registration handle if the callback function is successfully registered with the operating system. Otherwise, **KeRegisterProcessorChangeCallback** returns **NULL**. For more information about this handle, see the following Remarks section.

Remarks

A device driver calls the **KeRegisterProcessorChangeCallback** routine to register a callback function that is to be called by the operating system whenever a new processor is added to the hardware partition. When a user hot-plugs a new processor into the partition, the operating system calls the registered callback functions to rebalance the system resources that are allocated among the processors in the partition.

The registered callback function is called two times whenever a new processor is added to the hardware partition. The first time that the callback function is called, the **State** member of the KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains **KeProcessorAddStartNotify**. It is during this callback that the device driver should allocate any per-processor resources and perform any other tasks in preparation for the new processor to become active in the hardware partition. If an error occurs while the device driver processes this callback that could cause a bugcheck to occur if the operating system continues to add the new processor to the hardware partition, the callback function sets the variable that is pointed to by the *OperationStatus* parameter to an NTSTATUS code that describes the error condition. For example, if the device driver experiences a memory allocation failure for a per-processor data structure for a new processor, the callback function sets this variable to STATUS_INSUFFICIENT_RESOURCES. If the callback is processed successfully, the contents of the variable that is pointed to by the *OperationStatus* parameter should not be changed.

The second time that the callback function is called, the **State** member of the KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains either **KeProcessorAddCompleteNotify**, which indicates that the operating system has added the new processor to the hardware partition successfully, or **KeProcessorAddFailureNotify**, which indicates that the operating system did not add the new processor to the hardware partition. If the operating system successfully adds the new processor to the hardware partition, the callback function is not called the second time until after the new processor has been started and is available for thread scheduling. If the operating system did not add the new processor to the hardware partition, the device driver should free any per-processor resources during the second callback that it allocated during the first callback.

If the device driver specifies the KE_PROCESSOR_CHANGE_ADD_EXISTING flag in the *Flags* parameter when it calls the **KeRegisterProcessorChangeCallback** routine, the callback function is immediately called one time for each active processor that currently exists in the hardware partition. For these callbacks, the **State** member of the KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains **KeProcessorAddStartNotify**. It is during these callbacks that the device driver should allocate any per-processor resources and perform any other tasks to prepare to use the existing processors in the hardware partition. If the device driver successfully processes this callback for all active processors that currently exist in the hardware partition, the callback function is immediately called again for each active processor that currently exists in the hardware partition. For these callbacks, the **State** member of the

KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains **KeProcessorAddCompleteNotify**. **KeRegisterProcessorChangeCallback** returns after these callbacks complete.

If an error occurs while the device driver processes the first callback for one of the existing active processors in the hardware partition that could cause a bugcheck to occur if the driver were to continue to load, the callback function sets the variable pointed to by the *OperationStatus* parameter to an NTSTATUS code that describes the error condition. For example, if the device driver experiences a memory allocation failure for a per-processor data structure for an existing active processor, the callback function sets this variable to STATUS_INSUFFICIENT_RESOURCES. If the callback is processed successfully, the contents of the variable that is pointed to by the *OperationStatus* parameter should not be changed.

If the device driver indicates an error when the first callback for one of the existing active processors in the hardware partition is processed, the callback function is not called for any more of the existing active processors. Instead, the callback function is immediately called a second time for each active processor for which the callback was called the first time, excluding the active processor for which the callback indicated the error. For these callbacks, the **State** member of the KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT structure that is pointed to by the *ChangeContext* parameter contains **KeProcessorAddFailureNotify**.

A device driver typically calls the **KeRegisterProcessorChangeCallback** routine from within its DriverEntry routine. If the call to the **KeRegisterProcessorChangeCallback** routine returns **NULL**, the device driver's **DriverEntry** routine should return an NTSTATUS code that describes the error condition.

> **Note** A device driver can use the context that is passed in the *CallbackContext* parameter to the **KeRegisterProcessorChangeCallback** routine as a place where the callback function can store the NTSTATUS code that describes the error condition. This NTSTATUS code can then be used as the return value for the device driver's **DriverEntry** routine.

> **Note** The status value returned by **KeRegisterProcessorChangeCallback** indicates only whether the registration of the callback function succeeds or fails. It does not indicate the success or failure of any calls to callback functions that might occur before **KeRegisterProcessorChangeCallback** returns.

A callback function that has been registered for notification of processor changes must be unregistered before the device driver is unloaded from the operating system. To unregister the callback function, the device driver calls the KeDeregisterProcessorChangeCallback routine, and passes, as an input parameter to this routine, the registration handle that was returned by the call to the **KeRegisterProcessorChangeCallback** routine.

Requirements

Minimum supported client	Available in Windows Server 2008 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL

See Also

KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT

KeDeregisterProcessorChangeCallback

KeReleaseGuardedMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseGuardedMutex** routine releases a guarded mutex acquired with KeAcquireGuardedMutex or KeTryToAcquireGuardedMutex.

Syntax

```
void KeReleaseGuardedMutex(
  PKGUARDED_MUTEX Mutex
);
```

Parameters

`Mutex`

Pointer to the KGUARDED_MUTEX structure for the guarded mutex.

Return Value

None

Remarks

KeReleaseGuardedMutex implicitly exits the guarded region entered by **KeAcquireGuardedMutex** and **KeTryToAcquireGuardedMutex**.

For more information about guarded mutexes, see Fast Mutexes and Guarded Mutexes.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlKeApcLte1, HwStorPortProhibitedDDIs

See Also

KeAcquireGuardedMutex

KeReleaseGuardedMutexUnsafe function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseGuardedMutexUnsafe** routine releases a guarded mutex acquired by KeAcquireGuardedMutexUnsafe.

Syntax

```
void KeReleaseGuardedMutexUnsafe(
  PKGUARDED_MUTEX FastMutex
);
```

Parameters

`FastMutex`

Pointer to the KGUARDED_MUTEX structure for the guarded mutex.

Return Value

None

Remarks

For more information about guarded mutexes, see Fast Mutexes and Guarded Mutexes.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlKeApcLte1, HwStorPortProhibitedDDIs

See Also

KeAcquireGuardedMutexUnsafe

KeReleaseInStackQueuedSpinLock function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseInStackQueuedSpinLock** routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLock.

Syntax

```
void KeReleaseInStackQueuedSpinLock(
  PKLOCK_QUEUE_HANDLE LockHandle
);
```

Parameters

`LockHandle`

Specifies the queued spin lock handle initialized by the call to KeAcquireInStackQueuedSpinLock that acquired the lock.

Return Value

None

Remarks

This routine restores the original IRQL that the operating system saved at the beginning of the **KeAcquireInStackQueuedSpinLock** call.

For more information about queued spin locks, see Queued Spin Locks.

Use Hal.lib on 32-bit versions of Windows and Ntoskrnl.lib on 64-bit versions of Windows.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib; Ntoskrnl.lib
DLL	Hal.dll; Ntoskrnl.exe
IRQL	DISPATCH_LEVEL (see Remarks section)

DDI compliance rules	MarkingQueuedIrps, QueuedSpinLock, QueuedSpinLockRelease, HwStorPortProhibitedDDIs, QueuedSpinLock(storport), QueuedSpinLockRelease(storport)

See Also

KLOCK_QUEUE_HANDLE

KeAcquireInStackQueuedSpinLock

KeReleaseInStackQueuedSpinLockForDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseInStackQueuedSpinLockForDpc** routine releases a queued spin lock that was acquired by calling KeAcquireInStackQueuedSpinLockForDpc.

Syntax

```
void KeReleaseInStackQueuedSpinLockForDpc(
  PKLOCK_QUEUE_HANDLE LockHandle
);
```

Parameters

`LockHandle`

Pointer to the KLOCK_QUEUE_HANDLE structure that KeAcquireInStackQueuedSpinLockForDpc initialized when the queued spin lock was acquired.

Return Value

None

Remarks

For more information about queued spin locks, see Queued Spin Locks.

If the current IRQL was PASSIVE_LEVEL when the queued spin lock was acquired, **KeReleaseSpinLockForDpc** lowers the current IRQL to PASSIVE_LEVEL. If the current IRQL was DISPATCH_LEVEL when the queued spin lock was acquired, **KeReleaseSpinLockForDpc** does not change the current IRQL.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KLOCK_QUEUE_HANDLE

KeAcquireInStackQueuedSpinLockForDpc

KeReleaseInStackQueuedSpinLockFromDpcLevel function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseInStackQueuedSpinLockFromDpcLevel** routine releases a queued spin lock acquired by KeAcquireInStackQueuedSpinLockAtDpcLevel.

Syntax

```
void KeReleaseInStackQueuedSpinLockFromDpcLevel(
  PKLOCK_QUEUE_HANDLE LockHandle
);
```

Parameters

`LockHandle`

Specifies the queued spin lock handle initialized by the call to KeAcquireInStackQueuedSpinLockAtDpcLevel that acquired the lock.

Return Value

None

Remarks

For more information about queued spin locks, see Queued Spin Locks.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KLOCK_QUEUE_HANDLE

KeAcquireInStackQueuedSpinLockAtDpcLevel

KeReleaseInterruptSpinLock function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseInterruptSpinLock** routine releases an interrupt spin lock acquired by KeAcquireInterruptSpinLock.

Syntax

```
void KeReleaseInterruptSpinLock(
  PKINTERRUPT Interrupt,
  KIRQL       OldIrql
);
```

Parameters

`Interrupt`

Specifies the value of the *Interrupt* parameter passed to KeAcquireInterruptSpinLock.

`OldIrql`

Specifies the IRQL value returned by **KeAcquireInterruptSpinLock**.

Return Value

None

Remarks

The **KeReleaseInterruptSpinLock** releases the interrupt spin lock, and lowers the IRQL value back to the *OldIrql* value.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAcquireInterruptSpinLock

KeSynchronizeExecution

KeReleaseMutex function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseMutex** routine releases a mutex object, and specifies whether the caller is to call one of the **KeWait *Xxx*** routines as soon as **KeReleaseMutex** returns control.

Syntax

```
LONG KeReleaseMutex(
  PRKMUTEX Mutex,
  BOOLEAN  Wait
);
```

Parameters

`Mutex`

A pointer to an initialized mutex object for which the caller provides the storage.

`Wait`

Specifies whether the call to **KeReleaseMutex** is to be immediately followed by a call to one of the **KeWait*Xxx*** routines. If **TRUE**, the **KeReleaseMutex** call must be followed by a call to KeWaitForMultipleObjects, KeWaitForMutexObject, or KeWaitForSingleObject. For more information, see the following Remarks section.

Return Value

If the return value is zero, the mutex object was released and attained a state of *signaled*.

Remarks

For better performance, use fast mutexes or guarded mutexes. For more information, see Alternatives to Mutex Objects.

If the mutex object attains a signaled state, an attempt is made to satisfy a wait for the mutex object.

A mutex object can be released only by the thread that currently holds the mutex. If an attempt is made to release a mutex that the thread does not hold or if a mutex was acquired at IRQL = DISPATCH_LEVEL and the thread is not running at DISPATCH_LEVEL (and vice versa), the routine raises a STATUS_ABANDONED or STATUS_MUTEX_NOT_OWNED exception.

When a mutex object attains a signaled state, it is removed from the list of the mutexes that are held by that thread. If this list contains no more entries, the original priority for the thread is restored.

The **KeReleaseMutex** routine might temporarily raise the IRQL. If the *Wait* parameter is **FALSE**, the routine, before it returns, restores the IRQL to the original value that it had at the start of the call.

If *Wait* = **TRUE**, the routine returns without lowering the IRQL. In this case, the **KeReleaseMutex** call must be immediately followed by a **KeWait*Xxx*** call. By setting *Wait* = **TRUE**, the caller can prevent an unnecessary context switch from occurring between the **KeReleaseMutex** call and the **KeWait*Xxx*** call. The **KeWait*Xxx*** routine, before it returns, restores the IRQL to its original value at the start of the **KeReleaseMutex** call. Although the IRQL disables context switches between the two calls, these calls cannot reliably be used as the start and end of an

atomic operation. For example, between these two calls, a thread that is running at the same time on another processor might change the state of the event object or of the target of the wait.

If the caller is executing at IRQL = DISPATCH_LEVEL or in an arbitrary thread context, the *Timeout* parameter to **KeWait*Xxx*** must be zero.

If a mutex is acquired recursively, the holding thread must call **KeReleaseMutex** as many times as it acquired the mutex to set it to the signaled state.

For more information about mutex objects, see Mutex Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

ExReleaseFastMutex

ExReleaseFastMutexUnsafe

KeInitializeMutex

KeReadStateMutex

KeWaitForMultipleObjects

KeWaitForMutexObject

KeWaitForSingleObject

KeReleaseSemaphore function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseSemaphore** routine releases the specified semaphore object.

Syntax

```
LONG KeReleaseSemaphore(
  PRKSEMAPHORE Semaphore,
  KPRIORITY    Increment,
  LONG         Adjustment,
  BOOLEAN      Wait
);
```

Parameters

`Semaphore`

A pointer to an initialized semaphore object for which the caller provides the storage.

`Increment`

Specifies the priority increment to be applied if releasing the semaphore causes a wait to be satisfied.

`Adjustment`

Specifies a value to be added to the current semaphore count. This value must be positive.

`Wait`

Specifies whether the call to **KeReleaseSemaphore** is to be followed immediately by a call to one of the **KeWait*Xxx*** routines. If **TRUE**, the **KeReleaseSemaphore** call must be followed by a call to KeWaitForMultipleObjects, KeWaitForMutexObject, or KeWaitForSingleObject. For more information, see the following Remarks section.

Return Value

If the return value is zero, the previous state of the semaphore object is not-signaled.

Remarks

KeReleaseSemaphore supplies a run-time priority boost for waiting threads. If this call sets the semaphore to the signaled state, the semaphore count is augmented by the specified value. The caller can also specify whether it will call one of the **KeWait*Xxx*** routines as soon as **KeReleaseSemaphore** returns control.

Releasing a semaphore object causes the semaphore count to be augmented by the value of the *Adjustment* parameter. If the resulting value is greater than the limit of the semaphore object, the count is not adjusted and an exception, STATUS_SEMAPHORE_LIMIT_EXCEEDED, is raised.

Augmenting the semaphore object count causes the semaphore to attain a signaled state, and an attempt is made to satisfy as many waits as possible on the semaphore object.

The **KeReleaseSemaphore** routine might temporarily raise the IRQL. If the *Wait* parameter is **FALSE**, the routine, before it returns, restores the IRQL to the original value that it had at the start of the call.

If *Wait* = **TRUE**, the routine returns without lowering the IRQL. In this case, the **KeReleaseSemaphore** call must be immediately followed by a **KeWait*Xxx*** call. By setting *Wait* = **TRUE**, the caller can prevent an unnecessary context switch from occurring between the **KeReleaseSemaphore** call and the **KeWait*Xxx*** call. The **KeWait*Xxx*** routine, before it returns, restores the IRQL to its original value at the start of the **KeReleaseSemaphore** call. Although the IRQL disables context switches between the two calls, these calls cannot reliably be used as the start and end of an atomic operation. For example, between these two calls, a thread that is running at the same time on another processor might change the state of the semaphore object or of the target of the wait.

If the caller is executing at IRQL = DISPATCH_LEVEL or in an arbitrary thread context, the *Timeout* parameter to **KeWait*Xxx*** must be zero.

For more information about semaphore objects, see Semaphore Objects.

Callers of **KeReleaseSemaphore** must be running at IRQL <= DISPATCH_LEVEL provided that *Wait* is set to **FALSE**. Otherwise, the caller must be running at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeInitializeSemaphore

KeReadStateSemaphore

KeWaitForMultipleObjects

KeWaitForMutexObject

KeWaitForSingleObject

KeReleaseSpinLock macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseSpinLock** routine releases a spin lock and restores the original IRQL at which the caller was running.

Syntax

```
void KeReleaseSpinLock(
   a,
   b
);
```

Parameters

`a`

Pointer to a KSPIN_LOCK spin lock for which the caller provides the storage.

`b`

Specifies the KIRQL value saved from the preceding call to KeAcquireSpinLock.

Return Value

None

Remarks

This call is a reciprocal to KeAcquireSpinLock. The input *NewIrql* value must be the *OldIrql* returned by **KeAcquireSpinLock**.

For more information about spin locks, see Spin Locks.

Callers of this routine are running at IRQL = DISPATCH_LEVEL. On return from **KeReleaseSpinLock**, IRQL is restored to the *NewIrql* value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
IRQL	DISPATCH_LEVEL (see Remarks section)

DDI compliance rules	IrqlKeReleaseSpinLock, MarkingQueuedIrps, SpinLock, SpinLockDpc, SpinlockRelease, SpinLockSafe, ReqSendWhileSpinlock, Spinlock(kmdf), SpinlockDpc(kmdf), SpinlockRelease(kmdf), HwStorPortProhibitedDDIs, IrqlKeReleaseSpinLock(storport), SpinLock(storport), SpinLockDpc(storport), SpinLockRelease(storport), SpinLockSafe(storport)

See Also

KeAcquireSpinLock

KeInitializeSpinLock

KeReleaseSpinLockForDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseSpinLockForDpc** routine releases a spin lock that was acquired by calling KeAcquireSpinLockForDpc.

Syntax

```
void KeReleaseSpinLockForDpc(
  PKSPIN_LOCK SpinLock,
  KIRQL       OldIrql
);
```

Parameters

`SpinLock`

Pointer to the spin lock that **KeAcquireSpinLockForDpc** acquired.

`OldIrql`

Specifies the IRQL that was current when the spin lock was acquired. Specify the value returned by **KeAcquireSpinLockForDpc**.

Return Value

None

Remarks

For more information about spin locks, see Spin Locks.

If *OldIrql* = PASSIVE_LEVEL, **KeReleaseSpinLockForDpc** lowers the current IRQL to PASSIVE_LEVEL. If *OldIrql* = DISPATCH_LEVEL, **KeReleaseSpinLockForDpc** does not change the current IRQL.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAcquireSpinLockForDpc

KeReleaseSpinLockFromDpcLevel macro

1/25/2019 • 2 minutes to read • Edit Online

The **KeReleaseSpinLockFromDpcLevel** routine releases an executive spin lock without changing the IRQL.

Syntax

```
void KeReleaseSpinLockFromDpcLevel(
   a
);
```

Parameters

`a`

Pointer to an executive KSPIN_LOCK spin lock for which the caller provides the storage.

Return Value

None

Remarks

Drivers call **KeReleaseSpinLockFromDpcLevel** to release a spin lock acquired by calling KeAcquireSpinLockAtDpcLevel.

It is an error to call **KeReleaseSpinLockFromDpcLevel** if the specified spin lock was acquired by calling KeAcquireSpinLock because the caller's original IRQL is not restored, which can cause deadlocks or fatal page faults.

For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs, IrqlDispatch, SpinLockSafe

See Also

KeAcquireSpinLock

KeAcquireSpinLockAtDpcLevel

KeReleaseSpinLock

KeRemoveByKeyDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRemoveByKeyDeviceQueue** routine removes an entry, selected according to a sort key value, from the specified device queue.

Syntax

```
PKDEVICE_QUEUE_ENTRY KeRemoveByKeyDeviceQueue(
  PKDEVICE_QUEUE DeviceQueue,
  ULONG          SortKey
);
```

Parameters

`DeviceQueue`

Pointer to an initialized device queue object for which the caller provides the storage.

`SortKey`

Specifies the key to be used when searching the *DeviceQueue*.

Return Value

KeRemoveByKeyDeviceQueue returns the device queue entry that was removed; returns **NULL** if the queue was empty.

Remarks

This routine searches for the first entry in the device queue that has a value greater than or equal to the *SortKey*. After this entry is found, this routine removes the entry from the device queue and returns it. If no such entry is found, then the first entry in the queue is returned. If the device queue is empty, then the device is set to a not-busy state and a **NULL** pointer is returned.

It is an error to call **KeRemoveByKeyDeviceQueue** when the device queue object is set to a not-busy state.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, HwStorPortProhibitedDDIs, IrqlDispatch(storport)

See Also

KeInitializeDeviceQueue

KeInsertByKeyDeviceQueue

KeInsertDeviceQueue

KeRemoveDeviceQueue

KeRemoveEntryDeviceQueue

KeRemoveDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRemoveDeviceQueue** routine removes an entry from the head of a specified device queue.

Syntax

```
PKDEVICE_QUEUE_ENTRY KeRemoveDeviceQueue(
  PKDEVICE_QUEUE DeviceQueue
);
```

Parameters

`DeviceQueue`

Pointer to an initialized device queue object for which the caller provides the storage.

Return Value

If the device queue is empty but is set to a busy state, **KeRemoveDeviceQueue** returns **NULL**.

Remarks

The specified device queue spin lock is acquired and the state of the device queue is checked. If the device queue is set to a busy state and an IRP is queued, this routine dequeues the entry and returns a pointer to the IRP. A call to **KeRemoveDeviceQueue** when the device queue object is set to a busy state but no IRPs are queued causes a state change to not-busy. The specified device queue's spin lock is released.

It is an error to call **KeRemoveDeviceQueue** when the device queue object is set to a not-busy state.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, HwStorPortProhibitedDDIs, IrqlDispatch(storport)

See Also

KeInitializeDeviceQueue

KeInsertByKeyDeviceQueue

KeInsertDeviceQueue

KeRemoveByKeyDeviceQueue

KeRemoveEntryDeviceQueue

KeRemoveEntryDeviceQueue function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRemoveEntryDeviceQueue** routine returns whether the specified entry is in the device queue and removes it, if it was queued, from the device queue.

Syntax

```
BOOLEAN KeRemoveEntryDeviceQueue(
  PKDEVICE_QUEUE       DeviceQueue,
  PKDEVICE_QUEUE_ENTRY DeviceQueueEntry
);
```

Parameters

`DeviceQueue`

Pointer to an initialized device queue object for which the caller provides the storage.

`DeviceQueueEntry`

Pointer to the entry to be removed from the specified *DeviceQueue*.

Return Value

If the *DeviceQueueEntry* is queued, it is removed and **KeRemoveEntryDeviceQueue** returns **TRUE**.

Remarks

The IRQL is set to DISPATCH_LEVEL and the *DeviceQueue* spin lock is acquired.

If the specified *DeviceQueueEntry* is not in the queue, the IRP either is already being processed, or the IRP has been canceled. In this case, **KeRemoveEntryDeviceQueue** simply returns **FALSE**.

The specified *DeviceQueue* spin lock is released and IRQL is restored to its previous value.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeInitializeDeviceQueue

KeInsertByKeyDeviceQueue

KeInsertDeviceQueue

KeRemoveByKeyDeviceQueue

KeRemoveDeviceQueue

KeRemoveQueueDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRemoveQueueDpc** routine removes the specified DPC object from the system DPC queue.

Syntax

```
BOOLEAN KeRemoveQueueDpc(
  PRKDPC Dpc
);
```

Parameters

`Dpc`

A pointer to an initialized DPC object that was queued by a previous call to the KeInsertQueueDpc routine.

Return Value

KeRemoveQueueDpc returns **TRUE** if the DPC object is in the DPC queue. If the specified DPC object is not currently in the DPC queue, no operation is performed and **FALSE** is returned.

Remarks

If the specified DPC object is currently queued, it is removed from the queue, canceling a call to the associated DPC routine.

Starting with Windows Vista with Service Pack 1 (SP1) and Windows Server 2008, a return value of **TRUE** always means that **KeRemoveQueueDpc** successfully removed the DPC object from the DPC queue before the DPC routine started to run. In earlier versions of Windows, the DPC routine might occasionally run even if **KeRemoveQueueDpc** returns **TRUE**. In these earlier versions of Windows, drivers that cannot tolerate ambiguity in the **TRUE** return value should treat return values of **TRUE** and **FALSE** identically.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeInitializeDpc

KeInsertQueueDpc

KeResetEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KeResetEvent** routine resets a specified event object to a not-signaled state and returns the previous state of that event object.

Syntax

```
LONG KeResetEvent(
  PRKEVENT Event
);
```

Parameters

`Event`

A pointer to an initialized dispatcher object of type event for which the caller provides the storage.

Return Value

KeResetEvent returns a value that indicates the previous state of the specified *Event*, which is nonzero for a signaled state.

Remarks

Event is reset to a not-signaled state, meaning that its value is set to zero.

Unless the caller requires the value that is returned by **KeResetEvent**, using the KeClearEvent routine is a faster way to set an event object to a not-signaled state.

For more information about event objects, see Event Objects.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeClearEvent

KeInitializeEvent

KeReadStateEvent

KeSetEvent

KeWaitForMultipleObjects

KeWaitForSingleObject

KeRestoreExtendedProcessorState function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRestoreExtendedProcessorState** routine restores extended processor state information that was previously saved.

Syntax

```
void KeRestoreExtendedProcessorState(
  PXSTATE_SAVE XStateSave
);
```

Parameters

`XStateSave`

A pointer to an XSTATE_SAVE structure that contains the extended processor state information to restore. The contents of this structure must have been previously saved by the KeSaveExtendedProcessorState routine.

Return Value

None

Remarks

Kernel-mode driver code must ensure that calls to **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** are properly nested. This is required so that, at each nesting level, the state that was restored by the **KeRestoreExtendedProcessorState** call is the same state that was saved by the corresponding **KeSaveExtendedProcessorState** call. To ensure proper nesting, kernel-mode driver code must follow these rules:

- A **KeRestoreExtendedProcessorState** call that restores a saved state must be running at the same IRQL as the **KeSaveExtendedProcessorState** call that saved the state.
- If a pair of **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** calls is nested within a pair of surrounding **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** calls, the IRQL for the nested calls must not be lower than the IRQL for the surrounding calls.
- Typically, the caller-allocated **XSTATE_SAVE** structure that contains the state that was saved by **KeSaveExtendedProcessorState** resides on the stack. The stack naturally preserves the nesting of saved state information. If driver code stores the state in a location other than the stack, the driver writer must take special care to preserve the nesting of the **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** calls.
- The **KeRestoreExtendedProcessorState** call that restores a saved state must be running in the same thread as the **KeSaveExtendedProcessorState** call that saved the state.

A similar set of rules apply to the KeSaveFloatingPointState and KeRestoreFloatingPointState routines.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

KeRestoreFloatingPointState

KeSaveExtendedProcessorState

KeSaveFloatingPointState

XSTATE_SAVE

KeRestoreFloatingPointState function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRestoreFloatingPointState** routine restores the nonvolatile floating-point context saved by the preceding call to KeSaveFloatingPointState.

Syntax

```
NTSTATUS KeRestoreFloatingPointState(
  PKFLOATING_SAVE FloatSave
);
```

Parameters

`FloatSave`

A pointer to the KFLOATING_SAVE structure that was passed in the preceding call to KeSaveFloatingPointState.

Return Value

KeRestoreFloatingPointState returns STATUS_SUCCESS.

Remarks

KeRestoreFloatingPointState is the reciprocal of **KeSaveFloatingPointState**.

Any routine that calls **KeSaveFloatingPointState***must* call **KeRestoreFloatingPointState** before that routine returns control, and it must be running at the same IRQL as that from which the preceding call to **KeSaveFloatingPointState** occurred. Failure to meet either of these conditions causes a system bug check.

Kernel-mode driver code must ensure that calls to **KeSaveFloatingPointState** and **KeRestoreFloatingPointState** are properly nested. This is required so that, at each nesting level, the state that was restored by the **KeRestoreFloatingPointState** call is the same state that was saved by the corresponding **KeSaveFloatingPointState** call. To ensure proper nesting, kernel-mode driver code must follow these rules:

- A **KeRestoreFloatingPointState** call that restores a saved state must be running at the same IRQL as the **KeSaveFloatingPointState** call that saved the state.
- If a pair of **KeSaveFloatingPointState** and **KeRestoreFloatingPointState** calls is nested within a pair of surrounding **KeSaveFloatingPointState** and **KeRestoreFloatingPointState** calls, the IRQL for the nested calls must not be lower than the IRQL for the surrounding calls.
- Typically, the caller-allocated **KFLOATING_SAVE** structure that contains the state that was saved by **KeSaveFloatingPointState** resides on the stack. The stack naturally preserves the nesting of saved state information. If driver code stores the state in a location other than the stack, the driver writer must take special care to preserve the nesting of the **KeSaveFloatingPointState** and **KeRestoreFloatingPointState** calls.
- The **KeRestoreFloatingPointState** call that restores a saved state must be running in the same thread as the **KeSaveFloatingPointState** call that saved the state.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)"

See Also

KFLOATING_SAVE

KeSaveFloatingPointState

KeRevertToUserAffinityThreadEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeRevertToUserAffinityThreadEx** routine restores the previous affinity of the current thread.

Syntax

```
void KeRevertToUserAffinityThreadEx(
  KAFFINITY Affinity
);
```

Parameters

`Affinity`

A **KAFFINITY**-typed variable that specifies the new system affinity of the current thread. Set this parameter to the value that was returned by a previous call to the KeSetSystemAffinityThreadEx routine.

Return Value

None

Remarks

KeRevertToUserAffinityThreadEx changes the affinity of the current thread. The affinity value is a mask that identifies a set of processors on which the thread can run. If successful, the routine schedules the thread to run on a processor in this set.

If the *Affinity* value is nonzero, **KeRevertToUserAffinityThreadEx** sets the affinity mask of the thread to the *Affinity* value. If the *Affinity* value is zero, **KeRevertToUserAffinityThreadEx** restores the original user affinity mask of the thread. This original user affinity mask was saved in the thread object by a previous call to KeSetSystemAffinityThreadEx. A call to **KeRevertToUserAffinityThreadEx** has no effect unless it is preceded by a call to **KeSetSystemAffinityThreadEx**.

If the *Affinity* value is nonzero, the routine changes the affinity mask of the current thread to the *Affinity* value only if both of the following are true:

- The *Affinity* value is valid (that is, only mask bits that correspond to logical processors are set).
- At least one of the processors that is specified in the *Affinity* value is active.

If either of these conditions is not met, the call to **KeRevertToUserAffinityThreadEx** has no effect.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeRevertToUserGroupAffinityThread routine, which specifies a processor group, instead of **KeRevertToUserAffinityThreadEx**, which does not. However, the implementation of **KeRevertToUserAffinityThreadEx** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, a nonzero *Affinity* value is an affinity mask that specifies a set of logical processors in group 0 on which the thread can run. If *Affinity* is zero, the thread reverts to its original user-mode group number and affinity mask.

If **KeRevertToUserAffinityThreadEx** is called at IRQL <= APC_LEVEL and the call is successful, the new (reverted) affinity mask takes effect immediately. When the call returns, the calling thread is already running on a processor that is specified in the new affinity mask. If **KeRevertToUserAffinityThreadEx** is called at IRQL = DISPATCH_LEVEL and the call is successful, the pending processor change is deferred until the caller lowers the IRQL below DISPATCH_LEVEL.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)."
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KAFFINITY

KeRevertToUserGroupAffinityThread

KeSetSystemAffinityThreadEx

KeRevertToUserGroupAffinityThread function

1/25/2019 • 3 minutes to read • Edit Online

The **KeRevertToUserGroupAffinityThread** routine restores the group affinity of the calling thread to its original value at the time that the thread was created.

Syntax

```
void KeRevertToUserGroupAffinityThread(
  PGROUP_AFFINITY PreviousAffinity
);
```

Parameters

`PreviousAffinity`

A pointer to the group affinity to restore. This parameter points to a GROUP_AFFINITY structure that contains a group number and an affinity mask. The affinity mask specifies the set of logical processors that the user thread can run on.

Return Value

None

Remarks

This routine changes the group number and group-relative affinity mask of the calling thread. The group number and affinity mask identify a set of processors on which the thread can run. If successful, the routine schedules the thread to run on a processor in this set.

The *PreviousAffinity* parameter points to a **GROUP_AFFINITY** structure that specifies the new group number (**Group** member) and affinity mask (**Mask** member) for the thread. If *PreviousAffinity*->**Mask** is nonzero, **KeRevertToUserGroupAffinityThread** sets the group number and affinity mask of the calling thread to the values in the structure. If *PreviousAffinity*->**Mask** is zero, the routine restores the group number and affinity mask to their original values at the time that the thread was initialized.

A process can have affinity for more than one group at a time. However, a thread can be assigned to only one group at any time, and that group is always in the affinity of the thread's process.

A thread can change the group to which it is assigned by calling the KeSetSystemGroupAffinityThread routine. Following one or more calls to **KeSetSystemGroupAffinityThread**, the thread can restore the original group affinity that it had when the thread was created by calling **KeRevertToUserGroupAffinityThread**.

After the thread is created, a call to **KeRevertToUserGroupAffinityThread** has no effect (that is, the group number and affinity mask of the thread remain unchanged) unless the thread first calls **KeSetSystemGroupAffinityThread**. Following a call to **KeRevertToUserGroupAffinityThread**, a second call to **KeRevertToUserGroupAffinityThread** has no effect unless the thread first calls **KeSetSystemGroupAffinityThread**.

The routine changes the group number and affinity mask to the values that are specified in **PreviousAffinity* only if the following are true:

- The group number is valid.
- The affinity mask is valid (that is, only mask bits that correspond to logical processors in the group are set).
- At least one of the processors that is specified in the affinity mask is active.

If any of these conditions is not met, the call to **KeRevertToUserGroupAffinityThread** has no effect.

A related routine, KeRevertToUserAffinityThreadEx, changes the affinity mask of the calling thread, but this routine, unlike **KeRevertToUserGroupAffinityThread**, does not accept a group number as an input parameter. In Windows 7 and later versions of the Windows operating system, **KeRevertToUserAffinityThreadEx** assumes that the affinity mask refers to processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeRevertToUserAffinityThreadEx** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeRevertToUserGroupAffinityThread** instead of **KeRevertToUserAffinityThreadEx**.

If **KeRevertToUserGroupAffinityThread** is called at IRQL <= APC_LEVEL and the call is successful, the new (reverted) group affinity takes effect immediately. When the call returns, the calling thread is already running on a processor that is specified in the new group affinity. If **KeRevertToUserGroupAffinityThread** is called at IRQL = DISPATCH_LEVEL and the call is successful, the pending processor change is deferred until the caller lowers the IRQL below DISPATCH_LEVEL.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)."

See Also

GROUP_AFFINITY

KeRevertToUserAffinityThreadEx

KeSetSystemGroupAffinityThread

KeSaveExtendedProcessorState function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSaveExtendedProcessorState** routine saves extended processor state information.

Syntax

```
NTSTATUS KeSaveExtendedProcessorState(
  ULONG64      Mask,
  PXSTATE_SAVE XStateSave
);
```

Parameters

`Mask`

A 64-bit feature mask. The bits in this mask identify the extended processor feature states to save. If a mask bit is one, the routine saves the state of the feature that is identified by this bit. If a mask bit is zero, the state for the corresponding feature is not saved. This mask must not identify extended processor features that the operating system has not enabled. To obtain a mask of the enabled features, call the RtlGetEnabledExtendedFeatures routine.

A caller can set this parameter to the bitwise OR of one or more of the following **XSTATE_MASK_*XXX*** flag bits:

VALUE	MEANING
XSTATE_MASK_LEGACY_FLOATING_POINT	The floating-point extension (x87/MMX).
XSTATE_MASK_LEGACY_SSE	The streaming SIMD extension (SSE).

XST ATE _M ASK _LE GA CY	Both the x87/MMX and SSE extensions.
XST ATE _M ASK _GS SE	The Intel Sandy Bridge (formerly Gesher) SSE extension.

`XStateSave`

A pointer to a caller-allocated buffer into which the routine writes an XSTATE_SAVE structure. This structure contains the saved state information for the extended processor features indicated by the *Mask* parameter. The buffer must be large enough to contain this structure.

Return Value

KeSaveExtendedProcessorState returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STA TUS _IN SUF FICI ENT _RE SO URC ES	A memory allocation operation failed.

Remarks

On x86-based processors that support the XSAVE and XRSTOR instructions, these instructions provide a flexible mechanism to save and restore extended processor state information. **KeSaveExtendedProcessorState** uses these instructions if they are available.

To restore the extended processor state that was saved by **KeSaveExtendedProcessorState**, call the KeRestoreExtendedProcessorState routine.

The *Mask* parameter specifies the extended processor features whose state is to be saved. A **KeRestoreExtendedProcessorState** call restores only the extended processor state that was saved by the **KeSaveExtendedProcessorState** call that saved the state.

Kernel-mode code must save the state of an extended processor feature before it uses that feature, and it must restore the state before it exits.

Interrupt service routines (ISRs) run under severe time constraints that typically prevent them from using extended processor features. However, an ISR can schedule a deferred procedure call (DPC) that uses one or more extended processor features. The DPC routine must save and restore the state of the extended features to preserve the context of the interrupted program in whose process address space the routine runs.

The KeSaveFloatingPointState and KeRestoreFloatingPointState routines save and restore just the floating-point state (the x87/MMX registers) and the SSE state.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

KeRestoreExtendedProcessorState

KeRestoreFloatingPointState

KeSaveFloatingPointState

RtlGetEnabledExtendedFeatures

XSTATE_SAVE

KeSaveFloatingPointState function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSaveFloatingPointState** routine saves the nonvolatile floating-point context so the caller can carry out floating-point operations.

Syntax

```
NTSTATUS KeSaveFloatingPointState(
  PKFLOATING_SAVE FloatSave
);
```

Parameters

`FloatSave`

Pointer to a caller-allocated resident buffer, which must be at least **sizeof**(KFLOATING_SAVE).

Return Value

KeSaveFloatingPointState returns STATUS_SUCCESS if it saved the current thread's floating-point context and set up a fresh floating-point context for the caller. Otherwise, it returns one of the following error status codes.

RETURN CODE	DESCRIPTION
STATUS_ILLEGAL_FLOAT_CONTEXT	The system is configured to use floating-point emulation, rather than doing floating-point operations in the processors.
STATUS_INSUFFICIENT_RESOURCES	**KeSaveFloatingPointState** could not allocate sufficient memory to save the current thread's floating-point context.

Remarks

A successful call to **KeSaveFloatingPointState** allows the caller to carry out floating-point operations of its own,

but such a caller must restore the previous nonvolatile floating-point context as soon as its floating-point operations are done. Any routine that calls **KeSaveFloatingPointState** must call KeRestoreFloatingPointState before that routine returns control.

If the call to **KeSaveFloatingPointState** is successful, the data at *FloatSave* is opaque to the caller. The caller should not release the memory that it allocated for this data until after the call to KeRestoreFloatingPointState.

In Windows Vista and earlier versions of Windows, a **KeSaveFloatingPointState** call and the corresponding **KeRestoreFloatingPointState** call must occur in a guarded region. That is, the KeEnterGuardedRegion routine must be called before **KeSaveFloatingPointState** is called, and the KeLeaveGuardedRegion routine must be called after **KeRestoreFloatingPointState** is called. No such requirement exists in Windows 7 and later versions of Windows.

For performance reasons, drivers should avoid doing any floating-point operations unless absolutely necessary. The overhead of saving and restoring the current thread's nonvolatile floating-point state degrades the performance of any driver that does floating-point operations.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeEnterGuardedRegion

KeGetCurrentThread

KeLeaveGuardedRegion

KeRestoreFloatingPointState

PsCreateSystemThread

KeSetCoalescableTimer function

1/25/2019 • 5 minutes to read • Edit Online

The **KeSetCoalescableTimer** routine sets the initial expiration time and period of a timer object and specifies how much delay can be tolerated in the expiration times.

Syntax

```
BOOLEAN KeSetCoalescableTimer(
  PKTIMER       Timer,
  LARGE_INTEGER DueTime,
  ULONG         Period,
  ULONG         TolerableDelay,
  PKDPC         Dpc
);
```

Parameters

`Timer`

A pointer to a timer object. This parameter points to a KTIMER structure, which is an opaque, system structure that represents the timer object. This object must have been previously initialized by the KeInitializeTimerEx or KeInitializeTimer routine.

`DueTime`

Specifies an absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units, which are 100-nanosecond intervals. Absolute expiration times track any changes that are made to the system clock. Relative expiration times are not affected by system clock changes.

`Period`

Specifies the interval between periodic timer expirations in milliseconds. The value of this parameter must not exceed MAXLONG. This parameter is optional and can be set to zero to indicate that the timer is nonperiodic.

`TolerableDelay`

Specifies a tolerance, in milliseconds, for the timer period that *Period* specifies and for the initial time interval that *DueTime* specifies. For a periodic timer, the time interval between two successive timer expirations will be in the range from (*Period* - *TolerableDelay*) to (*Period* + *TolerableDelay*). The initial expiration time will be in the range from *DueTime* to (*DueTime* + *TolerableDelay*). The *TolerableDelay* value cannot be negative.

`Dpc`

A pointer to a DPC object. This parameter points to a KDPC structure, which is an opaque, system structure that represents the DPC object. This object must have been previously initialized by the KeInitializeDpc routine. This parameter is optional and can be specified as **NULL** if the caller does not require a DPC.

Return Value

KeSetCoalescableTimer returns **TRUE** if the timer object was already in the system timer queue. Otherwise, it

returns **FALSE**.

Remarks

This routine does the following:

- Sets the timer to a nonsignaled state.
- Associates the timer with the DPC, if a DPC is specified.
- Cancels the timer if it is already active.
- Makes the timer active and sets the due time and period of the timer to the specified values. The timer can expire immediately if the specified due time has already passed.

The KeSetTimerEx routine is similar to **KeSetCoalescableTimer** but does not accept a *TolerableDelay* parameter. The *TolerableDelay* parameter of **KeSetCoalescableTimer** enables the caller to specify a tolerance for the timer period. A call to **KeSetCoalescableTimer** with *TolerableDelay* = 0 is the same as a call to **KeSetTimerEx**. In many instances, developers can easily modify existing drivers by replacing calls to **KeSetTimerEx** with calls to **KeSetCoalescableTimer**.

If two **KeSetCoalescableTimer** calls specify the same timer object, and the second call occurs before the *DueTime* that is specified for the first call expires, the second call implicitly cancels the timer from the first call. However, if a timer expiration from the first call has already enabled a DPC to run, the DPC might run after the timer is canceled. The second call replaces the pending expiration time from the first call with a new expiration time, and activates the timer again.

If the *Period* parameter is nonzero, the timer is periodic. For a periodic timer, the first timer expiration occurs at the time indicated by the *DueTime* parameter. Later expirations are separated by the interval that is specified by *Period*. If *Period* = 0, the timer is nonperiodic and expires at the time that is indicated by *DueTime*.

If the *Dpc* parameter is non-**NULL**, the routine creates an association between the specified DPC object and the timer object. After the timer expires, the timer service removes the timer object from the system timer queue and sets this object to a signaled state. If a DPC object is associated with the timer object, the timer service inserts the DPC object into the system DPC queue to run as soon as conditions allow.

Only one instance of a particular DPC object can be in the system DPC queue at a time. To avoid potential race conditions, avoid passing the same DPC object to both the **KeSetCoalescableTimer** and KeInsertQueueDpc routines.

Avoid changing the importance or the target processor of a DPC that is associated with an active timer. Either cancel the timer or make sure that the timer has expired before you call a routine such as KeSetImportanceDpc or KeSetTargetProcessorDpcEx to change the DPC settings. For example, if a driver updates the target processor of a DPC while a timer enables the DPC to run, the DPC might run on an arbitrary processor.

A periodic timer automatically restarts as soon as it expires. Therefore, in a multiprocessor system, the DPC for a periodic timer might be running on two or more processors at the same time.

Drivers must cancel any active timers in their Unload routines. Call the KeCancelTimer routine to cancel a timer. If a DPC is associated with a timer that is periodic or that might recently have expired, a driver must wait (for example, by calling the KeFlushQueuedDpcs routine) to free the DPC and its associated data until all pending DPC executions on all processors finish.

KeSetCoalescableTimer uses the *TolerableDelay* parameter to perform timer coalescing. That is, the routine adjusts the expiration times for the timer to coincide with the expiration times of other software timers. Timer coalescing helps increase the length of idle periods so that the operating system can reduce power consumption and improve energy efficiency.

To use timer coalescing effectively, a caller should specify a *TolerableDelay* value of at least 32 milliseconds. This value equals two default system clock intervals of 15.6 milliseconds. If you can, use a larger *TolerableDelay* value, such as 100 milliseconds.

Try to specify the *Period* and *TolerableDelay* values in multiples of 50 milliseconds. Typical *Period* values are 50, 100, 250, 500, and 1000 milliseconds. Typical *TolerableDelay* values are 50, 100, 150, and 250 milliseconds.

Typically, a timer with a large *Period* value can use a proportionally large *TolerableDelay* value. For example, a timer with *Period* = 500 milliseconds might use *TolerableDelay* = 50 milliseconds, but a timer with *Period* = 10 seconds might use *TolerableDelay* = 1 second.

Expiration times are measured relative to the system clock, and the accuracy with which the operating system can detect when a timer expires is limited by the granularity of the system clock. For more information, see Timer Accuracy.

For more information about timer objects, see Timer Objects and DPCs. For more information about timer coalescing, see the Windows Timer Coalescing white paper on the WHDC website.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

KDPC

KTIMER

KeCancelTimer

KeFlushQueuedDpcs

KeInitializeDpc

KeInitializeTimer

KeInitializeTimerEx

KeInsertQueueDpc

KeSetImportanceDpc

KeSetTargetProcessorDpcEx

KeSetTimerEx

Unload

KeSetEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetEvent** routine sets an event object to a signaled state if the event was not already signaled, and returns the previous state of the event object.

Syntax

```
LONG KeSetEvent(
  PRKEVENT  Event,
  KPRIORITY Increment,
  BOOLEAN   Wait
);
```

Parameters

`Event`

A pointer to an initialized event object for which the caller provides the storage.

`Increment`

Specifies the priority increment to be applied if setting the event causes a wait to be satisfied.

`Wait`

Specifies whether the call to **KeSetEvent** is to be followed immediately by a call to one of the **KeWait*Xxx*** routines. If **TRUE**, the **KeSetEvent** call must be followed by a call to KeWaitForMultipleObjects, KeWaitForMutexObject, or KeWaitForSingleObject. For more information, see the following Remarks section.

Return Value

If the previous state of the event object was signaled, a nonzero value is returned.

Remarks

Calling **KeSetEvent** causes the event to attain a signaled state. If the event is a notification event, the system attempts to satisfy as many waits as possible on the event object. The event remains signaled until a call to KeClearEvent or KeResetEvent clears it. If the event is a synchronization event, one wait is satisfied before the system automatically clears the event.

The **KeSetEvent** routine might temporarily raise the IRQL. If the *Wait* parameter is **FALSE**, the routine, before it returns, restores the IRQL to the original value that it had at the start of the call.

If *Wait* = **TRUE**, the routine returns without lowering the IRQL. In this case, the **KeSetEvent** call must be immediately followed by a **KeWait*Xxx*** call. By setting *Wait* = **TRUE**, the caller can prevent an unnecessary context switch from occurring between the **KeSetEvent** call and the **KeWait*Xxx*** call. The **KeWait*Xxx*** routine, before it returns, restores the IRQL to its original value at the start of the **KeSetEvent** call. Although the IRQL disables context switches between the two calls, these calls cannot reliably be used as the start and end of an atomic operation. For example, between these two calls, a thread that is running at the same time on another processor might change the state of the event object or of the target of the wait.

A pageable thread or pageable driver routine that runs at IRQL = PASSIVE_LEVEL should never call **KeSetEvent** with the *Wait* parameter set to TRUE. Such a call causes a fatal page fault if the caller happens to be paged out between the calls to **KeSetEvent** and **KeWait*Xxx***.

For more information about event objects, see Event Objects.

If *Wait* is set to **FALSE**, the caller can be running at IRQL <= DISPATCH_LEVEL. Otherwise, callers of **KeSetEvent** must be running at IRQL <= APC_LEVEL and in a nonarbitrary thread context.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	CompletionEventChecking, IoAllocateIrpSignalEventInCompletion, IoBuildDeviceIoControlSetEvent, IoBuildFsdIrpSignalEventInCompletion, IrqlKeSetEvent, SignalEventInCompletion, DoubleKeSetEvent, HwStorPortProhibitedDDIs

See Also

KeClearEvent

KeInitializeEvent

KeReadStateEvent

KeResetEvent

KeWaitForMultipleObjects

KeWaitForMutexObject

KeWaitForSingleObject

Specifying Priority Boosts When Completing I/O Requests

KeSetImportanceDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetImportanceDpc** routine specifies how soon the DPC routine is run.

Syntax

```
void KeSetImportanceDpc(
  PRKDPC          Dpc,
  KDPC_IMPORTANCE Importance
);
```

Parameters

`Dpc`

Pointer to the caller's DPC object, which KeInitializeDpc already initialized.

`Importance`

Specifies one of the following system-defined values to determine the behavior of KeInsertQueueDpc and IoRequestDpc when either routine is used to queue the DPC.

LowImportance

Place the DPC at the end of the DPC queue, and do not begin processing of the queue.

MediumImportance

Place the DPC at the end of the DPC queue. If the DPC is assigned to the current processor's DPC queue, begin processing the queue immediately. **MediumImportance** is the default value for *Importance*.

MediumHighImportance

Place the DPC at the end of the DPC queue, and begin processing the queue immediately. MediumHighImportance is available only on Windows Vista and later versions of Windows.

HighImportance

Place the DPC at the beginning of the DPC queue, and begin processing the queue immediately.

Return Value

None

Remarks

The **KeSetImportanceDpc** routine influences how soon a DPC is run after it is queued by determining:

- The location of the DPC within the DPC queue. Typically, the **KeInsertQueueDpc** and **IoRequestDpc** routines place a DPC at the end of the queue. If a driver first calls **KeSetImportanceDpc** with *Importance* = **HighImportance**, **KeInsertQueueDpc** and **IoRequestDpc** will place the DPC at the beginning of the queue.
- When the system begins processing the DPC queue. Typically, **KeInsertQueueDpc** and **IoRequestDpc** immediately begin processing the DPC queue for the current processor. Drivers can specify different values for *Importance* to change this behavior.

By default, DPCs are assigned to the DPC queue for the current processor, so specifying **MediumImportance** or **MediumHighImportance** for *Importance* has the same effect. However, drivers can use KeSetTargetProcessorDpc to change the processor that the DPC will be assigned to.

For Windows Vista and later versions of the Windows operating system, you can use **KeSetImportanceDpc** for threaded DPCs. If the caller sets *Importance* to **HighImportance**, the DPC is placed at the beginning of the queue; otherwise, it is placed at the end. The routine does not affect when the threaded DPC queue is processed. Threaded DPCs are always processed by a dedicated thread at IRQL = PASSIVE_LEVEL. For more information about threaded DPCs, see Threaded DPCs.

Note that a driver must call **KeSetImportanceDpc** before it calls **KeInsertQueueDpc** and **IoRequestDpc** to have any effect.

For more information about how the system processes the DPC queue, see Organization of DPC Queues.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoRequestDpc

KeInitializeDpc

KeInsertQueueDpc

KeSetTargetProcessorDpc

KeSynchronizeExecution

KeSetPriorityThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetPriorityThread** routine sets the run-time priority of a driver-created thread.

Syntax

```
KPRIORITY KeSetPriorityThread(
  PKTHREAD  Thread,
  KPRIORITY Priority
);
```

Parameters

`Thread`

Pointer to the driver-created thread.

`Priority`

Specifies the priority of the driver-created thread, usually to the real-time priority value, LOW_REALTIME_PRIORITY. The value LOW_PRIORITY is reserved for system use.

Return Value

KeSetPriorityThread returns the old priority of the thread.

Remarks

If a call to **KeSetPriorityThread** resets the thread's priority to a lower value, execution of the thread can be rescheduled even if it is currently running or is about to be dispatched for execution.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

KeGetCurrentThread

KeQueryPriorityThread

KeSetBasePriorityThread

KeSetSystemAffinityThread function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetSystemAffinityThread** routine sets the system affinity of the current thread.

Syntax

```
void KeSetSystemAffinityThread(
  KAFFINITY Affinity
);
```

Parameters

`Affinity`

A **KAFFINITY**-typed variable that specifies the new system affinity of the current thread.

Return Value

None

Remarks

The implementation of **KeSetSystemAffinityThread** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeSetSystemAffinityThread** assigns the thread to group 0, and uses the affinity mask to specify a set of logical processors in this group on which the thread can run.

If **KeSetSystemAffinityThread** is called at IRQL <= APC_LEVEL and the call is successful, the new affinity mask takes effect immediately. When the call returns, the calling thread is already running on a processor that is specified in the new affinity mask. If **KeSetSystemAffinityThread** is called at IRQL = DISPATCH_LEVEL and the call is successful, the pending processor change is deferred until the caller lowers the IRQL below DISPATCH_LEVEL.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Use KeSetSystemAffinityThread only on systems before Windows Vista. In Windows Vista and later versions of the Windows operating system, use KeSetSystemAffinityThreadEx instead.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<= DISPATCH_LEVEL (see Remarks section)."
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KAFFINITY

KeSetSystemAffinityThreadEx

KeSetSystemAffinityThreadEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetSystemAffinityThreadEx** routine sets the system affinity of the current thread.

Syntax

```
KAFFINITY KeSetSystemAffinityThreadEx(
  KAFFINITY Affinity
);
```

Parameters

`Affinity`

A **KAFFINITY**-typed variable that specifies the new system affinity of the current thread.

Return Value

KeSetSystemAffinityThreadEx returns either the previous system affinity of the current thread, or zero to indicate that there was no previous system affinity.

Remarks

KeSetSystemAffinityThreadEx changes the affinity mask of the current thread. The affinity mask identifies a set of processors on which the thread can run. If successful, the routine schedules the thread to run on a processor in this set.

Callers of **KeSetSystemAffinityThreadEx** should save the return value and later pass this value to the **KeRevertToUserAffinityThreadEx** routine to restore the previous affinity mask.

In a multiprocessor system, a kernel-mode driver routine that runs in the context of a user-mode thread might need to call **KeSetSystemAffinityThreadEx** to temporarily change the affinity mask of the thread. Before the routine exits, it should call **KeRevertToUserAffinityThreadEx** to restore the affinity mask of the thread to its original value.

The term *user affinity* refers to the original affinity of the user-mode thread. The affinity that is set by the kernel-mode driver is referred to as the *system affinity*. If a call to **KeSetSystemAffinityThreadEx** replaces the user affinity of a thread with a system affinity, the call returns zero. Later, the driver restores the user affinity by calling **KeRevertToUserAffinityThreadEx** and passing this return value (zero) as the *Affinity* value.

Additionally, a kernel-mode driver routine that requires a particular affinity mask might call another kernel-mode routine that requires a different affinity mask. Each routine can call **KeSetSystemAffinityThreadEx** to set a new affinity mask and then call **KeRevertToUserAffinityThreadEx** to restore the previous affinity mask before returning.

The **KeSetSystemAffinityThreadEx** routine changes the affinity mask of the current thread to the *Affinity* value only if both of the following are true:

- The *Affinity* value is valid (that is, only mask bits that correspond to logical processors are set).
- At least one of the processors that is specified in the *Affinity* value is active.

If either of these conditions is not met, the call to **KeSetSystemAffinityThreadEx** has no effect.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeSetSystemGroupAffinityThread routine, which specifies a processor group, instead of **KeSetSystemAffinityThreadEx**, which does not. However, the implementation of **KeSetSystemAffinityThreadEx** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, **KeSetSystemAffinityThreadEx** assigns the thread to group 0, and uses the affinity mask to specify a set of logical processors in this group on which the thread can run. The routine returns the previous group-relative affinity mask, but not the previous group.

If **KeSetSystemAffinityThreadEx** is called at IRQL <= APC_LEVEL and the call is successful, the new affinity mask takes effect immediately. When the call returns, the calling thread is already running on a processor that is specified in the new affinity mask. If **KeSetSystemAffinityThreadEx** is called at IRQL = DISPATCH_LEVEL and the call is successful, the pending processor change is deferred until the caller lowers the IRQL below DISPATCH_LEVEL.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)."

See Also

KAFFINITY

KeRevertToUserAffinityThreadEx

KeSetSystemAffinityThread

KeSetSystemGroupAffinityThread

KeSetSystemGroupAffinityThread function

1/25/2019 • 5 minutes to read • Edit Online

The **KeSetSystemGroupAffinityThread** routine changes the group number and affinity mask of the calling thread.

Syntax

```
void KeSetSystemGroupAffinityThread(
  PGROUP_AFFINITY Affinity,
  PGROUP_AFFINITY PreviousAffinity
);
```

Parameters

`Affinity`

A pointer to a GROUP_AFFINITY structure that specifies the new group number and group-relative affinity mask for the calling thread.

`PreviousAffinity`

A pointer to a caller-allocated **GROUP_AFFINITY** structure into which the routine writes information about the previous group affinity of the calling thread. The caller can later use this pointer as an input parameter to the KeRevertToUserGroupAffinityThread routine to restore the previous thread affinity. Frequently, **KeSetSystemGroupAffinityThread** writes values to this structure that are not valid group affinities but that have special meaning to **KeRevertToUserGroupAffinityThread**. Do not supply pointers to these special values as *Affinity* parameters in subsequent **KeSetSystemGroupAffinityThread** calls.

If necessary, the caller can change the thread affinity more than once by calling **KeSetSystemGroupAffinityThread** multiple times. During the first of these calls, the caller should specify a non-**NULL** value for *PreviousAffinity* so that the original thread affinity can be captured and later restored. However, the later calls to **KeSetSystemGroupAffinityThread** can, as an option, set *PreviousAffinity* = **NULL**. For more information, see Remarks.

Return Value

None

Remarks

This routine changes the group number and group-relative affinity mask of the calling thread. The *Affinity* parameter points to a GROUP_AFFINITY structure. The group number and affinity mask in this structure identify a set of processors on which the thread can run. If successful, the routine schedules the thread to run on a processor in this set.

If the *PreviousAffinity* parameter is non-**NULL**, the routine saves information about the previous group affinity, which were in effect at the start of the call, in the **GROUP_AFFINITY** structure that *PreviousAffinity* points to. To restore the previous thread affinity, the caller can supply the pointer to this structure as an input parameter to the KeRevertToUserGroupAffinityThread routine.

In a multiprocessor system, a kernel-mode driver routine that runs in the context of a user-mode thread might need to call **KeSetSystemGroupAffinityThread** to temporarily change the group affinity of the thread. Before the routine exits, it should call **KeRevertToUserGroupAffinityThread** to restore the affinity mask of the thread to its original value.

A process can have affinity for more than one group at a time. However, a thread can be assigned to only one group at any time. That group is always in the affinity of the thread's process. A thread can change the group to which it is assigned by calling **KeSetSystemGroupAffinityThread**.

KeSetSystemGroupAffinityThread changes the group number and affinity mask to the values that are specified in **Affinity* only if the following are true:

- The group number is valid.
- The affinity mask is valid (that is, only mask bits that correspond to logical processors in the group are set).
- At least one of the processors that is specified in the affinity mask is active.

If any of these conditions is not met, the group number and affinity mask of the thread remain unchanged. If *PreviousAffinity* is non-**NULL**, the routine writes zero to both the group number and the affinity mask in **PreviousAffinity*.

Additionally, **KeSetSystemGroupAffinityThread** writes zero to both the group number and the affinity mask in **PreviousAffinity* if the previous group affinity was assigned to the thread in user mode. In response to a **GROUP_AFFINITY** structure in which the group number and affinity mask are both zero, **KeRevertToUserGroupAffinityThread** restores the current user-mode thread affinity. If the user-mode thread affinity changes between the **KeSetSystemGroupAffinityThread** and **KeRevertToUserGroupAffinityThread** calls, the most recent user-mode affinity is assigned to the thread. (An application can call a function such as SetThreadGroupAffinity to change the user-mode group affinity of a thread.)

Before the new affinity mask in **Affinity* takes effect, **KeSetSystemGroupAffinityThread** removes (sets to zero) any affinity mask bits that correspond to processors that are not currently active. In a subsequent **KeSetSystemGroupAffinityThread** call, the value that the routine writes to **PreviousAffinity* might contain an affinity mask that has been modified in this way.

A related routine, KeSetSystemAffinityThreadEx, changes the affinity mask of the calling thread, but this routine, unlike **KeSetSystemGroupAffinityThread**, does not accept a group number as an input parameter. Starting with Windows 7, **KeSetSystemAffinityThreadEx** assumes that the affinity mask refers to processors in group 0, which is compatible with the behavior of this routine in earlier versions of Windows that do not support groups. This behavior ensures that existing drivers that call **KeSetSystemAffinityThreadEx** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeSetSystemGroupAffinityThread** instead of **KeSetSystemAffinityThreadEx**.

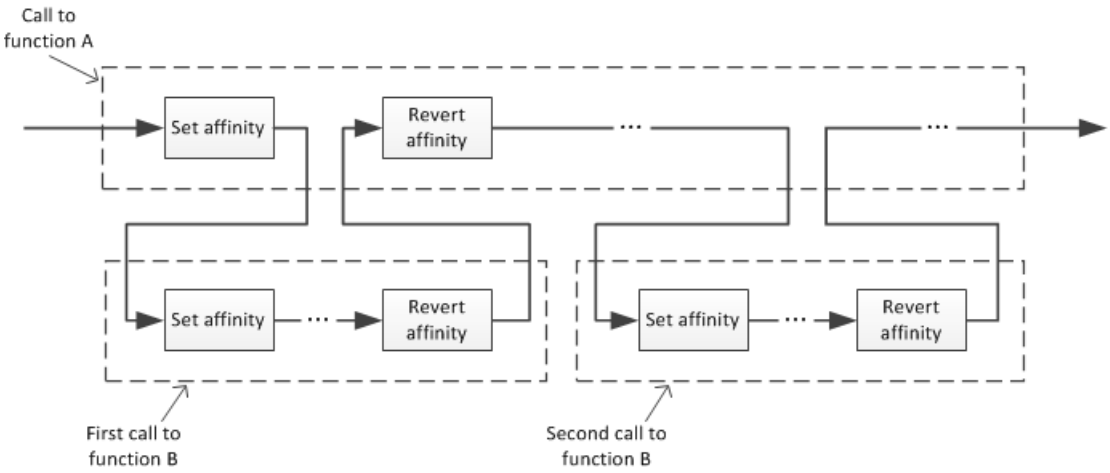
KeSetSystemGroupAffinityThread and **KeRevertToUserGroupAffinityThread** support a variety of calling patterns. Two examples are shown in the following diagrams.

The following diagram represents a driver thread that calls **KeSetSystemGroupAffinityThread** three times to change the thread affinity, and then calls **KeRevertToUserGroupAffinityThread** to restore the original thread affinity.

In the preceding diagram, the three boxes labeled "Set affinity" are calls to **KeSetSystemGroupAffinityThread**, and the box labeled "Revert affinity" is a call to **KeRevertToUserGroupAffinityThread**. The first

KeSetSystemGroupAffinityThread call uses the *PreviousAffinity* output pointer to save the original thread affinity. In the next two calls to **KeSetSystemGroupAffinityThread** (marked with asterisks), the caller sets *PreviousAffinity* to **NULL**. Before the thread exits, it calls **KeRevertToUserGroupAffinityThread** to restore the thread affinity that was saved by the first **KeSetSystemGroupAffinityThread** call.

The following diagram shows a somewhat different calling pattern in which pairs of **KeSetSystemGroupAffinityThread** and **KeRevertToUserGroupAffinityThread** calls are nested. In this diagram, each call to **KeSetSystemGroupAffinityThread** in the driver thread uses the *PreviousAffinity* output parameter to save the previous thread affinity, and each of these calls is paired with a call to **KeRevertToUserGroupAffinityThread** that restores the saved thread affinity.



In the preceding diagram, function A in the driver thread calls function B twice. Assume that on entry to function A, the thread still has the affinity assigned to it by the user-mode application. Thus, the **KeSetSystemGroupAffinityThread** call in function A saves the original, user-mode thread affinity. During the first call to function B, the **KeSetSystemGroupAffinityThread** saves the thread affinity assigned by the driver in function A, and the corresponding call to **KeRevertToUserGroupAffinityThread** restores this affinity. After B returns, the **KeRevertToUserGroupAffinityThread** in A restores the original, user-mode thread affinity. During the second call to B, the **KeSetSystemGroupAffinityThread** call saves the original, user-mode thread affinity, and the corresponding call to **KeRevertToUserGroupAffinityThread** restores this affinity. The point of this example is that function B does not need to know whether the caller (function A) changed the thread affinity to a driver-defined value before calling B.

If **KeSetSystemGroupAffinityThread** is called at IRQL <= APC_LEVEL and the call is successful, the new group affinity takes effect immediately. When the call returns, the calling thread is already running on a processor that is specified in the new group affinity. If **KeSetSystemGroupAffinityThread** is called at IRQL = DISPATCH_LEVEL and the call is successful, the pending processor change is deferred until the caller lowers the IRQL below DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Wdm.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (see Remarks section)."
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

GROUP_AFFINITY

KeRevertToUserGroupAffinityThread

KeSetSystemAffinityThreadEx

KeSetTargetProcessorDpc function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetTargetProcessorDpc** routine specifies the processor that a DPC routine will be run on.

Syntax

```
void KeSetTargetProcessorDpc(
  PRKDPC Dpc,
  CCHAR  Number
);
```

Parameters

`Dpc`

Pointer to the caller's DPC object, which KeInitializeDpc already initialized.

`Number`

Specifies the zero-based number of the target processor on which the DPC should be queued and executed.

Return Value

None

Remarks

On multiprocessor systems, each processor has its own DPC queue. The **KeSetTargetProcessorDpc** routine specifies which processor's queue the system should use when the driver calls KeInsertQueueDpc or IoRequestDpc to queue a DPC to be run later.

Starting with Windows Vista, you can also use **KeSetTargetProcessorDpc** to specify the target processor for threaded DPCs.

A call to **KeSetTargetProcessorDpcEx** that occurs after a DPC object has been queued has no effect on the selection of a processor for the DPC routine to run on. To control the selection of the target processor, a **KeSetTargetProcessorDpc** call must occur before the call to **KeInsertQueueDpc** or **IoRequestDpc** that queues the DPC object.

For more information about DPC queues, see Organization of DPC Queues.

Windows 7 and later versions of Windows support processor groups. Drivers that are designed to handle information about processor groups should use the KeSetTargetProcessorDpcEx routine, which specifies a processor group, instead of **KeSetTargetProcessorDpc**, which does not. However, the implementation of **KeSetTargetProcessorDpc** in Windows 7 and later versions of Windows provides compatibility for drivers that were written for earlier versions of Windows, which do not support processor groups. In this implementation, if *Number* is less than the number of active logical processors in group 0, **KeSetTargetProcessorDpc** sets the target for the DPC to the processor in group 0 that is specified by *Number*. Otherwise, the DPC target does not change.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoRequestDpc

KeGetCurrentProcessorNumber

KeInitializeDpc

KeInsertQueueDpc

KeSetImportanceDpc

KeSetTargetProcessorDpcEx

KeSetTargetProcessorDpcEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetTargetProcessorDpcEx** routine specifies the processor that a DPC routine will run on.

Syntax

```
NTSTATUS KeSetTargetProcessorDpcEx(
  PKDPC             Dpc,
  PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`Dpc`

A pointer to the caller's DPC object. This parameter points to a KDPC structure, which is an opaque, system structure that represents the DPC object. This object must previously have been initialized by the KeInitializeDpc routine.

`ProcNumber`

A pointer to a caller-allocated PROCESSOR_NUMBER structure that identifies the target processor on which the DPC will be queued and executed. This structure specifies a group and a processor within this group.

Return Value

KeSetTargetProcessorDpcEx returns STATUS_SUCCESS if the call is successful. Otherwise, it returns the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ProcNumber* parameter points to an invalid processor number.

Remarks

Each processor in a multiprocessor system has its own DPC queue. **KeSetTargetProcessorDpcEx** specifies which processor's queue the system should use when the driver calls the KeInsertQueueDpc or IoRequestDpc routine to queue a DPC to be run later.

KeSetTargetProcessorDpcEx can specify the target processor for both ordinary DPCs and threaded DPCs. An ordinary DPC cannot be preempted by even a high-priority thread, but a threaded DPC can be preempted by time-critical threads that have sufficiently high priorities.

A related routine, KeSetTargetProcessorDpc, specifies a target processor for a DPC, but this routine, unlike **KeSetTargetProcessorDpcEx**, does not specify a group for the target processor. Starting with Windows 7, **KeSetTargetProcessorDpc** assumes that the target processor belongs to group 0. This behavior ensures that existing drivers that call **KeSetTargetProcessorDpc** and that use no group-oriented features will run correctly in multiprocessor systems that have two or more groups. However, drivers that use any group-oriented features in Windows 7 and later versions of the Windows operating system should call **KeSetTargetProcessorDpcEx** instead of **KeSetTargetProcessorDpc**.

A call to **KeSetTargetProcessorDpcEx** that occurs after a DPC object has been queued has no effect on the selection of a processor for the DPC routine to run on. To control the selection of the target processor, a **KeSetTargetProcessorDpcEx** call must occur before the call to **KeInsertQueueDpc** or **IoRequestDpc** that queues the DPC object.

For more information about DPC queues, see Organization of DPC Queues.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IoRequestDpc

KDPC

KeInitializeDpc

KeInsertQueueDpc

KeSetTargetProcessorDpc

PROCESSOR_NUMBER

KeSetTimer function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetTimer** routine sets the absolute or relative interval at which a timer object is to be set to a signaled state and, optionally, supplies a CustomTimerDpc routine to be executed when that interval expires.

Syntax

```
BOOLEAN KeSetTimer(
  PKTIMER       Timer,
  LARGE_INTEGER DueTime,
  PKDPC         Dpc
);
```

Parameters

`Timer`

Pointer to a timer object that was initialized with KeInitializeTimer or KeInitializeTimerEx.

`DueTime`

Specifies the absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

`Dpc`

Pointer to a DPC object that was initialized by KeInitializeDpc. This parameter is optional.

Return Value

If the timer object was already in the system timer queue, **KeSetTimer** returns **TRUE**.

Remarks

The **KeSetTimer** routine does the following:

- Computes the expiration time.
- Sets the timer to a not-signaled state.
- Inserts the timer object in the system timer queue.

If the timer object was already in the timer queue, it is implicitly canceled before being set to the new expiration time. A call to **KeSetTimer** before the previously specified *DueTime* has expired cancels both the timer and the call to the *Dpc*, if any, associated with the previous call.

If the *Dpc* parameter is specified, a DPC object is associated with the timer object. When the timer expires, the timer object is removed from the system timer queue and its state is set to signaled. If a DPC object was associated with the timer when it was set, the DPC object is inserted in the system DPC queue to be executed as soon as conditions permit after the timer interval expires.

Expiration times are measured relative to the system clock, and the accuracy with which the operating system can detect when a timer expires is limited by the granularity of the system clock. For more information, see Timer Accuracy.

Only one instantiation of a given DPC object can be queued at any given moment. To avoid potential race conditions, the DPC passed to **KeSetTimer** should not be passed to KeInsertQueueDpc.

Drivers must cancel any active timers in their Unload routines. Use KeCancelTimer to cancel any timers.

Callers of **KeSetTimer** can specify one expiration time for a timer. To set a recurring timer use KeSetTimerEx.

For more information about timer objects, see Timer Objects and DPCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeCancelTimer

KeInitializeDpc

KeInitializeTimer

KeInitializeTimerEx

KeReadStateTimer

KeSetTimerEx

KeWaitForMultipleObjects

KeWaitForSingleObject

KeSetTimerEx function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSetTimerEx** routine sets the absolute or relative interval at which a timer object is to be set to a signaled state, optionally supplies a CustomTimerDpc routine to be executed when that interval expires, and optionally supplies a recurring interval for the timer.

Syntax

```
BOOLEAN KeSetTimerEx(
  PKTIMER       Timer,
  LARGE_INTEGER DueTime,
  LONG          Period,
  PKDPC         Dpc
);
```

Parameters

`Timer`

Pointer to a timer object that was initialized with KeInitializeTimer or KeInitializeTimerEx.

`DueTime`

Specifies the absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

`Period`

Specifies an optional recurring interval for the timer in milliseconds. Must be a value that is greater than or equal to zero. If the value of this parameter is zero, the timer is a nonperiodic timer that does not automatically re-queue itself.

`Dpc`

Pointer to a DPC object that was initialized by KeInitializeDpc. This parameter is optional.

Return Value

If the timer object was already in the system timer queue, **KeSetTimerEx** returns **TRUE**.

Remarks

The **KeSetTimerEx** routine does the following:

- Computes the expiration time.
- Sets the timer to a not-signaled state.
- Sets the recurring interval for the timer, if one was specified.
- Inserts the timer object in the system timer queue.

If the timer object was already in the timer queue, it is implicitly canceled before being set to the new expiration time. A call to **KeSetTimerEx** before the previously specified *DueTime* has expired cancels both the timer and the call to the *Dpc*, if any, associated with the previous call.

Expiration times are measured relative to the system clock, and the accuracy with which the operating system can detect when a timer expires is limited by the granularity of the system clock. For more information, see Timer Accuracy.

If the *Dpc* parameter is specified, a DPC object is associated with the timer object. When the timer expires, the timer object is removed from the system timer queue and it is set to a signaled state. If a DPC object was associated with the timer when it was set, the DPC object is inserted in the system DPC queue to be executed as soon as conditions permit after the timer interval expires.

A DPC routine cannot deallocate a periodic timer. A DPC routine can deallocate a nonperiodic timer.

Note that a periodic timer is automatically restarted as soon as it expires. Thus, on a multiprocessor machine, the DPC for a periodic timer can be running on two processors simultaneously.

Only one instantiation of a given DPC object can be queued at any given moment. To avoid potential race conditions, the DPC passed to **KeSetTimerEx** should not be passed to KeInsertQueueDpc.

Drivers must cancel any active timers in their Unload routines. Use KeCancelTimer to cancel any timers.

For more information about timer objects, see Timer Objects and DPCs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlKeDispatchLte, HwStorPortProhibitedDDIs

See Also

KeCancelTimer

KeInitializeDpc

KeInitializeTimer

KeInitializeTimerEx

KeReadStateTimer

KeWaitForMultipleObjects

KeWaitForSingleObject

KeStallExecutionProcessor function

1/25/2019 • 2 minutes to read • Edit Online

The **KeStallExecutionProcessor** routine stalls the caller on the current processor for a specified time interval.

Syntax

```
NTHALAPI VOID KeStallExecutionProcessor(
  ULONG MicroSeconds
);
```

Parameters

`MicroSeconds`

Specifies the number of microseconds to stall.

Return Value

None

Remarks

KeStallExecutionProcessor is a processor-dependent routine that busy-waits for at least the specified number of microseconds, but not significantly longer.

This routine is for use by device drivers and other software that must wait for an interval of less than a clock tick but more than for a few instructions. If you use this routine you must minimize the stall interval, typically to less than 50 microseconds. If a driver must wait for a longer interval, you should use a different synchronization technique.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Hal.lib
DLL	Hal.dll
IRQL	Any level

See Also

KeDelayExecutionThread

KeWaitForMultipleObjects

KeWaitForSingleObject

KeSynchronizeExecution function

1/25/2019 • 2 minutes to read • Edit Online

The **KeSynchronizeExecution** routine synchronizes the execution of the specified routine with the interrupt service routine (ISR) that is assigned to a set of one or more interrupt objects.

Syntax

```
BOOLEAN KeSynchronizeExecution(
  PKINTERRUPT            Interrupt,
  PKSYNCHRONIZE_ROUTINE  SynchronizeRoutine,
  __drv_aliasesMem PVOID SynchronizeContext
);
```

Parameters

`Interrupt`

A pointer to a set of interrupt objects. The caller obtained this pointer from the IoConnectInterrupt or IoConnectInterruptEx routine.

`SynchronizeRoutine`

Specifies a caller-supplied SynchCritSection routine whose execution is to be synchronized with the execution of the ISR assigned to the interrupt objects.

`SynchronizeContext`

A pointer to a caller-supplied context value to be passed to the SynchCritSection routine when it is called.

Return Value

KeSynchronizeExecution returns **TRUE** if the operation succeeds. Otherwise, it returns **FALSE**.

Remarks

When this routine is called, the following occurs:

1. The IRQL is raised to the *SynchronizeIrql* value specified in the call to IoConnectInterrupt or IoConnectInterruptEx.
2. Access to *SynchronizeContext* is synchronized with the assigned ISR by acquiring the associated interrupt object spin lock (or system event object, in the case of an ISR that runs at PASSIVE_LEVEL).
3. The specified SynchCritSection routine is called with the *SynchronizeContext* value as its parameter.

If the ISR runs at DIRQL >= DISPATCH_LEVEL, the *SynchCritSection* routine runs at the same DIRQL and must therefore run for as brief a time as possible to avoid delaying other high-priority tasks.

Callers of **KeSynchronizeExecution** must be running at IRQL <= DIRQL; that is, at an IRQL that is less than or equal to the value of the **SynchronizeIrql** value that the caller specified when it registered its ISR with **IoConnectInterrupt** or **IoConnectInterruptEx**.

Starting with Windows 8, a driver can call **KeSynchronizeExecution** to synchronize execution of a *SynchCritSection* routine with an ISR that runs at IRQL = PASSIVE_LEVEL. In earlier versions of Windows, **KeSynchronizeExecution** can synchronize execution only with an ISR that runs at IRQL >= DISPATCH_LEVEL. For more information, see Using Passive-Level Interrupt Service Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DIRQL (see Remarks section)"

See Also

IoConnectInterrupt

IoConnectInterruptEx

KeTestSpinLock function

1/25/2019 • 2 minutes to read • Edit Online

The **KeTestSpinLock** routine tests for the availability of a spin lock.

Syntax

```
BOOLEAN KeTestSpinLock(
  PKSPIN_LOCK SpinLock
);
```

Parameters

`SpinLock`

A pointer to a spin lock. The spin lock must have been initialized by calling KeInitializeSpinLock.

Return Value

KeTestSpinLock returns **FALSE** if the spin lock is currently being held. Otherwise, it returns **TRUE**.

Remarks

This routine enables the caller that is running at a low IRQL to test the spin lock. If **KeTestSpinLock** returns **TRUE**, the caller can try to acquire the spin lock with a high probability of immediate success.

This routine neither raises nor lowers the IRQL, and it never acquires the spin lock.

For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

KeAcquireSpinLock

KeInitializeSpinLock

KeReleaseSpinLock

KeTryToAcquireSpinLockAtDpcLevel function

1/25/2019 • 2 minutes to read • Edit Online

The **KeTryToAcquireSpinLockAtDpcLevel** routine attempts to acquire a spin lock at DISPATCH_LEVEL.

Syntax

```
BOOLEAN KeTryToAcquireSpinLockAtDpcLevel(
  PKSPIN_LOCK SpinLock
);
```

Parameters

`SpinLock`

Specifies the spin lock to acquire. The spin lock must have already been initialized by KeInitializeSpinLock.

Return Value

KeTryToAcquireSpinLockAtDpcLevel returns **TRUE** if the spin lock has been acquired, and **FALSE** if the spin lock is already being held and cannot be acquired.

Remarks

If the specified spin lock is not busy, the **KeTryToAcquireSpinLockAtDpcLevel** routine acquires the spin lock (see KeAcquireSpinLock for details) and returns **TRUE**. If the spin lock has already been acquired, the routine immediately returns **FALSE**.

If the spin lock is acquired, the caller can release it by using the KeReleaseSpinLock routine.

If you want the driver to block when it is unable to acquire the spin lock, use KeAcquireSpinLockAtDpcLevel instead.

For more information about spin locks, see Spin Locks.

Requirements

Minimum supported client	Available in Windows Server 2003 with Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	DISPATCH_LEVEL
DDI compliance rules	SpinLock, SpinlockRelease, HwStorPortProhibitedDDIs

See Also

KeAcquireSpinLock

KeAcquireSpinLockAtDpcLevel

KeInitializeSpinLock

KeReleaseSpinLock

KeWaitForMultipleObjects function

1/25/2019 • 6 minutes to read • Edit Online

The **KeWaitForMultipleObjects** routine puts the current thread into an alertable or nonalertable wait state until any or all of a number of dispatcher objects are set to a signaled state or (optionally) until the wait times out.

Syntax

```
NTSTATUS KeWaitForMultipleObjects(
  ULONG                                                                  Count,
  PVOID []                                                               Object,
  WAIT_TYPE                                                              WaitType,
  KWAIT_REASON                                                           WaitReason,
  __drv_strictType(KPROCESSOR_MODE / enum _MODE,__drv_typeConst)KPROCESSOR_MODE WaitMode,
  BOOLEAN                                                                Alertable,
  PLARGE_INTEGER                                                         Timeout,
  PKWAIT_BLOCK                                                           WaitBlockArray
);
```

Parameters

`Count`

The number of objects to be waited on. This parameter specifies the number of elements in the array pointed to by the *Object* parameter.

`Object`

A pointer to an array of pointers to dispatcher objects (events, mutexes, semaphores, threads, and timers) for which the caller supplies the storage. Both the pointer array and the dispatcher objects must reside in nonpaged system memory. For more information, see Remarks.

`WaitType`

The type of wait operation to perform. Specify either **WaitAll**, indicating that all of the specified objects must attain a signaled state before the wait is satisfied; or **WaitAny**, indicating that any one of the objects must attain a signaled state before the wait is satisfied.

`WaitReason`

The reason for the wait. Drivers should set this value to **Executive** or, if the driver is doing work on behalf of a user and is running in the context of a user thread, to **UserRequest**.

`WaitMode`

Whether the caller waits in **KernelMode** or **UserMode**. Intermediate and lowest-level drivers should specify **KernelMode**. If the set of objects waited on includes a mutex, the caller must specify **KernelMode**.

`Alertable`

A Boolean value that indicates whether the thread can be alerted while it is in the waiting state.

`Timeout`

A pointer to a time-out value that specifies the absolute or relative time, in 100-nanosecond units, at which the wait

is to be completed.

A positive value specifies an absolute time, relative to January 1, 1601. A negative value specifies an interval relative to the current time. Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

If **Timeout* = 0, the routine returns without waiting. If the caller supplies a **NULL** pointer, the routine waits indefinitely until any or all of the dispatcher objects are set to the signaled state. For more information, see the following Remarks section.

`WaitBlockArray`

A pointer to a caller-allocated **KWAIT_BLOCK** array. If *Count* <= THREAD_WAIT_OBJECTS, then *WaitBlockArray* can be **NULL**. Otherwise, this parameter must point to a memory buffer of **sizeof**(**KWAIT_BLOCK**) * *Count* bytes. The routine uses this buffer for record-keeping while performing the wait operation. The *WaitBlockArray* buffer must reside in nonpaged system memory. For more information, see Remarks.

Return Value

KeWaitForMultipleObjects can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The caller specified **WaitAll** for the *WaitType* parameter and all dispatcher objects in the *Object* array have been set to the signaled state.
STATUS_ALERTED	The wait was interrupted to deliver an alert to the calling thread.
STATUS_USER_APC	The wait was interrupted to deliver a user asynchronous procedure call (APC) to the calling thread.
STATUS_TIMEOUT	A time-out occurred before the specified set of wait conditions was met. This value can be returned when an explicit time-out value of zero is specified, but the specified set of wait conditions cannot be met immediately.

Note that the NT_SUCCESS macro recognizes all of these status values as "success" values.

Remarks

Each thread object has a built-in array of wait blocks that can be used to wait for several objects to be set concurrently. Whenever possible, the built-in array of wait blocks should be used in a wait-multiple operation

because no additional wait block storage needs to be allocated and later deallocated. However, if the number of objects that must be waited on concurrently is greater than the number of built-in wait blocks, use the *WaitBlockArray* parameter to specify an alternate set of wait blocks to be used in the wait operation. Drivers only need to allocate a sufficiently large memory buffer for *WaitBlockArray*. The buffer does not need to be initialized; however, it must be allocated from nonpaged system memory. If the *WaitMode* parameter is **UserMode**, the *WaitBlockArray* buffer must not be allocated on the local stack because the stack might be swapped out of memory. Drivers can treat this buffer as an opaque structure and can free it after the routine returns. If either *Count* > MAXIMUM_WAIT_OBJECTS or if *WaitBlockArray* is **NULL** and *Count* > THREAD_WAIT_OBJECTS, the system issues Bug Check 0xC (MAXIMUM_WAIT_OBJECTS_EXCEEDED).

The current state for each of the specified objects is examined to determine whether the wait can be satisfied immediately. If the necessary side effects are performed on the objects, an appropriate value is returned.

If the wait cannot be satisfied immediately and either no time-out value or a nonzero time-out value has been specified, the current thread is put in a waiting state and a new thread is selected for execution on the current processor. If no *Timeout* is supplied, the calling thread will remain in a wait state until the conditions specified by *Object* and *WaitType* are satisfied.

If *Timeout* is specified, the wait will be automatically satisfied if none of the specified wait conditions is met when the given interval expires.

A time-out value of zero allows the testing of a set of wait conditions, conditionally performing any side effects if the wait can be immediately satisfied, as in the acquisition of a mutex.

Time-out intervals are measured relative to the system clock, and the accuracy with which the operating system can detect the end of a time-out interval is limited by the granularity of the system clock. For more information, see Timer Accuracy.

The *Alertable* parameter determines when the thread can be alerted and its wait state consequently aborted. For additional information, see Waits and APCs.

The array pointed to by the *Objects* parameter must reside in nonpaged system memory. Typically, a driver allocates the storage for the *Objects* array on the local stack. The *Objects* array can be allocated on the local stack regardless of the value of the *WaitMode* parameter.

The dispatcher objects pointed to by the elements in the *Objects* array must reside in nonpaged system memory. If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must never attempt to pass parameters on the stack when calling **KeWaitForMultipleObjects** with the **UserMode** argument. If you allocate the event on the stack, you must set the *WaitMode* parameter to **KernelMode**.

A special consideration applies when the *Object* parameter passed to **KeWaitForMultipleObjects** is a mutex. If the dispatcher object waited on is a mutex, APC delivery is the same as for all other dispatcher objects during the wait. However, after **KeWaitForMultipleObjects** returns with STATUS_SUCCESS and the thread actually holds the mutex, only special kernel-mode APCs are delivered. Delivery of all other APCs, both kernel-mode and user-mode, is disabled. This restriction on the delivery of APCs persists until the mutex is released.

It is especially important to check the return value of **KeWaitForMultipleObjects** when the *WaitMode* parameter is **UserMode** or *Alertable* is **TRUE**, because **KeWaitForMultipleObjects** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to **FALSE**.

Where possible, *Alertable* should be set to **FALSE** and *WaitMode* should be set to **KernelMode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long-term wait.

A mutex can be recursively acquired only MINLONG times. If this limit is exceeded, the routine raises a

STATUS_MUTANT_LIMIT_EXCEEDED exception.

Callers of **KeWaitForMultipleObjects** can be running at IRQL <= DISPATCH_LEVEL. However, if *Timeout* = **NULL** or **Timeout* != 0, the caller must be running at IRQL <= APC_LEVEL and in a nonarbitrary thread context. (If *Timeout* != **NULL** and **Timeout* = 0, the caller must be running at IRQL <= DISPATCH_LEVEL.)

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrpProcessingComplete, IrqlKeWaitForMultipleObjects, HwStorPortProhibitedDDIs, SpNoWait

See Also

ExInitializeFastMutex

KeInitializeEvent

KeInitializeMutex

KeInitializeSemaphore

KeInitializeTimer

KeWaitForMutexObject

KeWaitForSingleObject

KeWaitForSingleObject function

1/25/2019 • 5 minutes to read • Edit Online

The **KeWaitForSingleObject** routine puts the current thread into a wait state until the given dispatcher object is set to a signaled state or (optionally) until the wait times out.

Syntax

```
NTSTATUS KeWaitForSingleObject(
  PVOID                                                                Object,
  KWAIT_REASON                                                         WaitReason,
  __drv_strictType(KPROCESSOR_MODE / enum _MODE,__drv_typeConst)KPROCESSOR_MODE WaitMode,
  BOOLEAN                                                              Alertable,
  PLARGE_INTEGER                                                       Timeout
);
```

Parameters

`Object`

Pointer to an initialized dispatcher object (event, mutex, semaphore, thread, or timer) for which the caller supplies the storage.

`WaitReason`

Specifies the reason for the wait. A driver should set this value to **Executive**, unless it is doing work on behalf of a user and is running in the context of a user thread, in which case it should set this value to **UserRequest**.

`WaitMode`

Specifies whether the caller waits in **KernelMode** or **UserMode**. Lowest-level and intermediate drivers should specify **KernelMode**. If the given *Object* is a mutex, the caller must specify **KernelMode**.

`Alertable`

Specifies a Boolean value that is **TRUE** if the wait is alertable and **FALSE** otherwise.

`Timeout`

Pointer to a time-out value that specifies the absolute or relative time, in 100-nanosecond units, at which the wait is to be completed.

A positive value specifies an absolute time, relative to January 1, 1601. A negative value specifies an interval relative to the current time. Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

If **Timeout* = 0, the routine returns without waiting. If the caller supplies a **NULL** pointer, the routine waits indefinitely until the dispatcher object is set to the signaled state. For more information, see the following Remarks section.

Return Value

KeWaitForSingleObject can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The dispatcher object specified by the *Object* parameter satisfied the wait.
STATUS_ALERTED	The wait was interrupted to deliver an alert to the calling thread.
STATUS_USER_APC	The wait was interrupted to deliver a user asynchronous procedure call (APC) to the calling thread.
STATUS_TIMEOUT	A time-out occurred before the object was set to a signaled state. This value can be returned when the specified set of wait conditions cannot be immediately met and *Timeout* is set to zero.

Note that the NT_SUCCESS macro recognizes all of these status values as "success" values.

Remarks

The current state of the specified *Object* is examined to determine whether the wait can be satisfied immediately. If so, the necessary side effects are performed on the object. Otherwise, the current thread is put in a waiting state and a new thread is selected for execution on the current processor.

The *Alertable* parameter determines when the thread can be alerted and its wait state consequently aborted. For additional information, see Waits and APCs.

A special consideration applies when the *Object* parameter passed to **KeWaitForSingleObject** is a mutex. If the dispatcher object waited on is a mutex, APC delivery is the same as for all other dispatcher objects during the wait. However, after **KeWaitForSingleObject** returns with STATUS_SUCCESS and the thread actually holds the mutex, only special kernel-mode APCs are delivered. Delivery of all other APCs, both kernel-mode and user-mode, is disabled. This restriction on the delivery of APCs persists until the mutex is released.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must never attempt to pass parameters on the stack when calling **KeWaitForSingleObject** using the **UserMode** argument. If you allocate the event on the stack, you must set the *WaitMode* parameter to **KernelMode**.

It is especially important to check the return value of **KeWaitForSingleObject** when the *WaitMode* parameter is **UserMode** or *Alertable* is **TRUE**, because **KeWaitForSingleObject** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long-term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to **FALSE**.

Where possible, *Alertable* should be set to **FALSE** and *WaitMode* should be set to **KernelMode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long-term wait.

If a **NULL** pointer is supplied for *Timeout*, the calling thread remains in a wait state until the *Object* is signaled.

A time-out value of zero allows the testing of a set of wait conditions and for the conditional performance of any side effects if the wait can be immediately satisfied, as in the acquisition of a mutex.

Time-out intervals are measured relative to the system clock, and the accuracy with which the operating system can detect the end of a time-out interval is limited by the granularity of the system clock. For more information, see Timer Accuracy.

A mutex can be recursively acquired only MINLONG times. If this limit is exceeded, the routine raises a STATUS_MUTANT_LIMIT_EXCEEDED exception.

Callers of **KeWaitForSingleObject** must be running at IRQL <= DISPATCH_LEVEL. However, if *Timeout* = **NULL** or **Timeout* != 0, the caller must be running at IRQL <= APC_LEVEL and in a nonarbitrary thread context. (If *Timeout* != **NULL** and **Timeout* = 0, the caller must be running at IRQL <= DISPATCH_LEVEL.)

KeWaitForMutexObject is a macro that converts to **KeWaitForSingleObject**, which can be used instead.

For better performance, use fast mutexes or guarded mutexes. For more information, see Alternatives to Mutex Objects.

For more information about mutex objects, see Mutex Objects.

A mutex can be recursively acquired only MINLONG times. If this limit is exceeded, the routine raises a STATUS_MUTANT_LIMIT_EXCEEDED exception.

Callers of **KeWaitForMutexObject** must be running at IRQL <= DISPATCH_LEVEL. However, if *Timeout* = **NULL** or **Timeout* != 0, the caller must be running at IRQL <= APC_LEVEL and in a nonarbitrary thread context. (If *Timeout* != **NULL** and **Timeout* = 0, the caller must be running at IRQL <= DISPATCH_LEVEL.)

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

DDI compliance rules	CompleteRequestStatusCheck, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildFsdIrpSignalEventInCompletionTimeout, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, IrpProcessingComplete, IrqlKeWaitForMutexObject, LowerDriverReturn, MarkIrpPending2, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControlInternal, RemoveLockForwardRead, RemoveLockForwardWrite, StartDeviceWait, StartDeviceWait2, StartDeviceWait3, StartDeviceWait4, HwStorPortProhibitedDDIs, SpNoWait

See Also

ExInitializeFastMutex

KeBugCheckEx

KeInitializeEvent

KeInitializeMutex

KeInitializeSemaphore

KeInitializeTimer

KeWaitForMultipleObjects

KIPI_BROADCAST_WORKER callback function

1/25/2019 • 2 minutes to read • Edit Online

The *IpiGenericCall* routine runs simultaneously on all processors.

Syntax

```
KIPI_BROADCAST_WORKER KipiBroadcastWorker;

ULONG_PTR KipiBroadcastWorker(
  ULONG_PTR Argument
)
{...}
```

Parameters

`Argument`

Supplies the value that was passed to the KeIpiGenericCall routine that called *IpiGenericCall*.

Return Value

IpiGenericCall returns a driver-defined value. If *IpiGenericCall* ran on the same processor that called **KeIpiGenericCall**, **KeIpiGenericCall** returns the driver-defined value that *IpiGenericCall* returns. Otherwise, the value is ignored.

Remarks

IpiGenericCall routines run at IRQL = IPI_LEVEL, which is greater than DIRQL for every device. *IpiGenericCall* routines must satisfy the same restrictions as bug check callback routines. For more information about these restrictions, see Writing a Bug Check Callback Routine.

Examples

To define an *IpiGenericCall* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *IpiGenericCall* callback routine that is named `MyIpiGenericCall`, use the KIPI_BROADCAST_WORKER type as shown in this code example:

```
KIPI_BROADCAST_WORKER MyIpiGenericCall;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
ULONG_PTR
  MyIpiGenericCall(
    ULONG_PTR  Argument
    )
  {
      // Function body
  }
```

The KIPI_BROADCAST_WORKER function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KIPI_BROADCAST_WORKER function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IPI_LEVEL.

See Also

KeIpiGenericCall

KMESSAGE_SERVICE_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

An *InterruptMessageService* routine services a message-signaled interrupt.

Syntax

```
KMESSAGE_SERVICE_ROUTINE KmessageServiceRoutine;

BOOLEAN KmessageServiceRoutine(
  _KINTERRUPT *Interrupt,
  PVOID ServiceContext,
  ULONG MessageID
)
{...}
```

Parameters

`Interrupt`

A pointer to the KINTERRUPT structure for the interrupt. The driver received this pointer in the call to the IoConnectInterruptEx routine that registered the driver's *InterruptMessageService* routine.

`ServiceContext`

The *ServiceContext* value that the driver passed to IoConnectInterruptEx when the *InterruptMessageService* routine was registered.

`MessageID`

Return Value

The *InterruptMessageService* routine returns **TRUE** if the interrupt is one handled by the *InterruptMessageService* routine. Otherwise, it returns **FALSE**.

Remarks

Drivers use IoConnectInterruptEx to register an *InterruptMessageService* routine to handle their message-signaled interrupts. A driver can subsequently unregister the routine by calling IoDisconnectInterruptEx. Message-signaled interrupts are supported starting with Windows Vista.

The system can call an *InterruptMessageService* routine even when the routine's interrupt has not occurred. For example, if a message-signaled interrupt is shared, *InterruptMessageService* can be called for interrupts belonging to other devices. The routine must check whether the value for the *ServiceContext* parameter matches the value passed to **IoConnectInterruptEx**. If the value does match, *InterruptMessageService* handles the interrupt and returns **TRUE**. Otherwise, *InterruptMessageService* does not handle the interrupt and returns **FALSE**.

Note that if the system receives multiple identical interrupts over a short time interval, it can combine these into a single call to *InterruptMessageService*. The routine must be written to handle multiple identical interrupts within a single call.

Message-signaled interrupts are similar in behavior to edge-triggered interrupts. The device sends an interrupt

request but does not receive any hardware acknowledgment that the request was received.

An *InterruptMessageService* executes at an IRQL greater than or equal to the maximum device IRQL (DIRQL) for every interrupt the routine handles.

Examples

To define an *InterruptMessageService* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *InterruptMessageService* callback routine that is named `MyInterruptMessageService`, use the KMESSAGE_SERVICE_ROUTINE type as shown in this code example:

```
KMESSAGE_SERVICE_ROUTINE MyInterruptMessageService;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
BOOLEAN
  MyInterruptMessageService(
    struct _KINTERRUPT  *Interrupt,
    PVOID  ServiceContext,
    ULONG  MessageId
    )
  {
      // Function body
  }
```

The KMESSAGE_SERVICE_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KMESSAGE_SERVICE_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	See Remarks section.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	See Remarks section.

See Also

IO_INTERRUPT_MESSAGE_INFO

IoConnectInterruptEx

IoDisconnectInterruptEx

KSERVICE_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterruptService* routine (ISR) quickly services a device interrupt and schedules post-interrupt processing of received data, if necessary.

Syntax

```
KSERVICE_ROUTINE KserviceRoutine;

BOOLEAN KserviceRoutine(
  _KINTERRUPT *Interrupt,
  PVOID ServiceContext
)
{...}
```

Parameters

`Interrupt`

Caller-supplied pointer to the KINTERRUPT structure for the interrupt.

`ServiceContext`

Caller-supplied pointer to context information, specified in a previous call to IoConnectInterrupt or IoConnectInterruptEx.

Return Value

If the routine determines that the interrupt did not come from one of the driver's devices, it must return **FALSE**. Otherwise, the routine must service the interrupt and return **TRUE**.

Remarks

To register an ISR for a specific interrupt vector and processor affinity, a driver must call IoConnectInterrupt or IoConnectInterruptEx.

A driver's *InterruptService* routine (ISR) executes in an interrupt context, at some system-assigned DIRQL, as specified by the *SynchronizeIrql* parameter to **IoConnectInterrupt**. (Other devices, with higher DIRQL values, can interrupt the ISR.)

Before the system calls an ISR, it acquires the interrupt's spin lock (the *SpinLock* parameter to **IoConnectInterrupt**), so the ISR cannot simultaneously execute on another processor. After the ISR returns, the system releases the spin lock.

An ISR must first determine if the interrupt came from one of the driver's devices, by examining context information supplied by *Context*. If the interrupt is not from one of the driver's devices, the routine must immediately return **FALSE** so the I/O manager can call other drivers that have registered ISRs for the same processor and interrupt vector.

For more information about implementing ISRs, see Interrupt Service Routines.

Examples

To define an *InterruptService* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define an *InterruptService* callback routine that is named `MyInterruptService`, use the KSERVICE_ROUTINE type as shown in this code example:

```
KSERVICE_ROUTINE MyInterruptService;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
BOOLEAN
  MyInterruptService(
    struct _KINTERRUPT  *Interrupt,
    PVOID  ServiceContext
    )
  {
      // Function body
  }
```

The KSERVICE_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KSERVICE_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DIRQL (see Remarks section).

KSTART_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ThreadStart* routine provides an entry point for a driver-created system thread.

Syntax

```
KSTART_ROUTINE KstartRoutine;

void KstartRoutine(
  PVOID StartContext
)
{...}
```

Parameters

`StartContext`

A caller-supplied pointer to driver-defined context information that is specified in the *StartContext* parameter a previous call to PsCreateSystemThread.

Return Value

None

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

PsCreateSystemThread

KSYNCHRONIZE_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *SynchCritSection* routine is used to access hardware resources or driver data that are shared with a driver's InterruptService routine.

Syntax

```
KSYNCHRONIZE_ROUTINE KsynchronizeRoutine;

BOOLEAN KsynchronizeRoutine(
  PVOID SynchronizeContext
)
{...}
```

Parameters

`SynchronizeContext`

Caller-supplied context information, specified by the driver's call to KeSynchronizeExecution.

Return Value

If the routine's operation succeeds, the routine should return **TRUE**; otherwise, it should return **FALSE**. (Success and failure of this routine are driver-defined.) The specified return value becomes the return value for **KeSynchronizeExecution**.

Remarks

Drivers must use *SynchCritSection* routines to access hardware resources or driver data that can also be accessed by an *InterruptService* routine (ISR).

The system calls a driver's *SynchCritSection* routine when the driver calls KeSynchronizeExecution. When a driver calls **KeSynchronizeExecution**, it specifies the address of a *SynchCritSection* routine, context information for the routine, and an interrupt object pointer. The **KeSynchronizeExecution** routine acquires the interrupt object's spin lock, then calls the *SynchCritSection* routine.

A driver's *SynchCritSection* routine executes at the same IRQL as the ISR with which it is associated. Specifically, it executes at some system-assigned DIRQL, as specified by the *SynchronizeIrql* parameter to IoConnectInterrupt. (Other devices, with higher DIRQL values, can interrupt a *SynchCritSection* routine.)

Examples

To define a *SynchCritSection* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *SynchCritSection* callback routine that is named `MySynchCritSection`, use the KSYNCHRONIZE_ROUTINE type as shown in this code example:

```
KSYNCHRONIZE_ROUTINE MySynchCritSection;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
BOOLEAN
  MySynchCritSection(
    PVOID  SynchronizeContext
    )
  {
      // Function body
  }
```

The KSYNCHRONIZE_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the KSYNCHRONIZE_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at DIRQL (see Remarks section).

MM_MDL_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

A driver-supplied callback routine that is invoked after a memory descriptor list (MDL) is mapped by calling the MmMapMdl function.

Syntax

```
MM_MDL_ROUTINE MmMdlRoutine;

void MmMdlRoutine(
  PVOID DriverContext,
  PVOID MappedVa
)
{...}
```

Parameters

`DriverContext`

A pointer to a driver-defined context. The driver's callback function can store any status information in the driver context and then examine the value, when the callback is invoked.

`MappedVa`

A pointer to a buffer that contains the system virtual address of the mapping.

Return Value

This callback function does not return a value.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h
IRQL	"<=DISPATCH_LEVEL"

See Also

MmMapMdl

MmAdvanceMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **MmAdvanceMdl** routine advances the beginning of an MDL's virtual memory range by the specified number of bytes.

Syntax

```
NTSTATUS MmAdvanceMdl(
  PMDL  Mdl,
  ULONG NumberOfBytes
);
```

Parameters

`Mdl`

Specifies the MDL to advance.

`NumberOfBytes`

Specifies the number of bytes to advance the beginning of the MDL.

Return Value

MmAdvanceMdl returns an NTSTATUS code. The possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully advanced the beginning of the MDL.
STATUS_INVALID_PARAMETER_2	The caller attempted to advance the beginning of the MDL past the end.

Remarks

MmAdvanceMdl advances only the beginning of the virtual memory address range. The ending address remains the same, and the length of the range is shrunk accordingly.

A higher-level driver can use **MmAdvanceMdl** under low-memory conditions when a lower-level driver can only partially complete a read/write request. The higher-level driver can use **MmAdvanceMdl** to advance past the part of the buffer that has already been read or written, and then reissue the IRP to complete the request. (The driver can, of course, repeat this process as many times as necessary.)

If **MmAdvanceMdl** advances past the initial page, any pages that **MmAdvanceMdl** passed are immediately unlocked, and the system virtual address that maps the MDL and the user address are also adjusted.

Use of **MmAdvanceMdl** can slow system performance. It must be used only when all of the following conditions hold:

- The higher-level driver, in its own I/O handling, can only complete certain I/O requests after transferring a fixed amount of data, but the lower-level driver only transfers data in smaller amounts. (An example is a network transport driver for the SPX or NBT protocols. Each protocol supports reliable message passing for messages that are bigger than one Ethernet frame. The transport driver can only complete a read request for such a message once it has reassembled the message from multiple Ethernet frames.)
- The higher-level driver already tried and failed to allocate a new MDL to transfer a data fragment from an incomplete I/O request. (If the driver succeeds in allocating a new MDL, it must use that MDL and IoBuildPartialMdl to perform the I/O request instead of **MmAdvanceMdl**.)
- The higher-level driver must continue to make progress, even under low-memory conditions.

Drivers that do not satisfy these conditions must instead use the **IoBuildPartialMdl** routine to complete any partially-successful I/O operations.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IoBuildPartialMdl

MmAllocateContiguousMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmAllocateContiguousMemory** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemory(
  SIZE_T           NumberOfBytes,
  PHYSICAL_ADDRESS HighestAcceptableAddress
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF. If you do not have specific requirements for memory allocation, set to MAXULONG64.

Return Value

MmAllocateContiguousMemory returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

MmAllocateContiguousMemory allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemory** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemory** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemory**.

Note When physical memory is fragmented on a computer that has a large amount of RAM, calls to **MmAllocateContiguousMemory**, which require the operating system to search for contiguous blocks of memory, can severely degrade performance. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemory**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

Memory that **MmAllocateContiguousMemory** allocates is uninitialized. A kernel-mode driver must first set this memory to zero if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

AllocateCommonBuffer

DriverEntry

MmAllocateContiguousMemorySpecifyCache

MmAllocateNonCachedMemory

MmFreeContiguousMemory

MmAllocateContiguousMemorySpecifyCache function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocateContiguousMemorySpecifyCache** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemorySpecifyCache(
  SIZE_T              NumberOfBytes,
  PHYSICAL_ADDRESS    LowestAcceptableAddress,
  PHYSICAL_ADDRESS    HighestAcceptableAddress,
  PHYSICAL_ADDRESS    BoundaryAddressMultiple,
  MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching allowed for the requested memory.

Return Value

MmAllocateContiguousMemorySpecifyCache returns the base virtual address for the allocated memory. If the system is unable to allocate the requested buffer, the routine returns **NULL**.

Remarks

MmAllocateContiguousMemorySpecifyCache allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemorySpecifyCache** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemorySpecifyCache** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemorySpecifyCache**.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** If you use the **MmAllocateContiguousMemorySpecifyCache** routine on computers with large amounts of memory, the operating system's performance might severely degrade when the system tries to create a contiguous chunk of memory. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemorySpecifyCache**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

> **Note** Memory that **MmAllocateContiguousMemorySpecifyCache** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemory

MmFreeContiguousMemory

MmAllocateContiguousMemorySpecifyCacheNode function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocateContiguousMemorySpecifyCacheNode** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousMemorySpecifyCacheNode(
  SIZE_T              NumberOfBytes,
  PHYSICAL_ADDRESS    LowestAcceptableAddress,
  PHYSICAL_ADDRESS    HighestAcceptableAddress,
  PHYSICAL_ADDRESS    BoundaryAddressMultiple,
  MEMORY_CACHING_TYPE CacheType,
  NODE_REQUIREMENT    PreferredNode
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching requested for the contiguous physical memory.

`PreferredNode`

The preferred node number. If a multiprocessor system contains N nodes, the nodes are numbered 0 to N-1. If MM_ANY_NODE_OK is specified or the machine only has one node, then the allocation is satisfied from any node. Otherwise, the allocation is made from the preferred node or if a satisfactory range cannot be found from the preferred node, then **NULL** is returned.

Return Value

MmAllocateContiguousMemorySpecifyCacheNode returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

A kernel-mode device driver calls this routine to allocate a contiguous block of physical memory. In a non-uniform memory access (NUMA) multiprocessor system, the caller can specify a preferred node from which to allocate the memory. A node is a collection of processors that share fast access to a region of memory. In a non-NUMA multiprocessor or a single-processor system, **MmAllocateContiguousMemorySpecifyCacheNode** treats all memory as belonging to a single node and allocates memory from this node.

MmAllocateContiguousMemorySpecifyCacheNode allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousMemorySpecifyCacheNode** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousMemorySpecifyCacheNode** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousMemorySpecifyCacheNode**.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** If you use the **MmAllocateContiguousMemorySpecifyCacheNode** routine on computers with large amounts of memory, the operating system's performance might severely degrade when the system tries to create a contiguous chunk of memory. This degradation is greatly reduced starting with Windows Vista SP1 and Windows Server 2008, but contiguous memory can still be expensive to allocate. For this reason, drivers should avoid repeated calls to **MmAllocateContiguousMemorySpecifyCacheNode**. Instead, drivers should allocate all required contiguous buffers in their **DriverEntry** routines and reuse these buffers.

> **Note** Memory that **MmAllocateContiguousMemorySpecifyCacheNode** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows Vista.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DriverEntry

MEMORY_CACHING_TYPE

MmFreeContiguousMemory

MmAllocateContiguousNodeMemory function

1/25/2019 • 4 minutes to read • Edit Online

The **MmAllocateContiguousNodeMemory** routine allocates a range of contiguous, nonpaged physical memory and maps it to the system address space.

Syntax

```
PVOID MmAllocateContiguousNodeMemory(
  SIZE_T           NumberOfBytes,
  PHYSICAL_ADDRESS LowestAcceptableAddress,
  PHYSICAL_ADDRESS HighestAcceptableAddress,
  PHYSICAL_ADDRESS BoundaryAddressMultiple,
  ULONG            Protect,
  NODE_REQUIREMENT PreferredNode
);
```

Parameters

`NumberOfBytes`

The size, in bytes, of the block of contiguous memory to allocate. For more information, see Remarks.

`LowestAcceptableAddress`

The lowest valid physical address the caller can use. For example, if a device can address only locations above the first 8 megabytes of the processor's physical memory address range, the driver for this device should set *LowestAcceptableAddress* to 0x0000000000800000.

`HighestAcceptableAddress`

The highest valid physical address the caller can use. For example, if a device can address only locations in the first 16 megabytes of the processor's physical memory address range, the driver for this device should set *HighestAcceptableAddress* to 0x0000000000FFFFFF.

`BoundaryAddressMultiple`

The physical address multiple that the allocated buffer must not cross. A physical address multiple must always be a power of two. This parameter is optional and can be specified as zero to indicate that the device has no special memory boundary restrictions. For more information, see Remarks.

`Protect`

Flag bits that specify the protection to use for the allocated memory. The caller must set one (but not both) of the following flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_READWRITE	Allocate read/write, no-execute (NX) memory. Most callers should set this flag bit. For more information, see Remarks.

PAGE_EXECUTE_READWRITE	Allocate read/write memory that is executable. This flag bit should be set only if the caller requires the ability to execute instructions in the allocated memory.

In addition, the caller can set one (but not both) of the following optional flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_NOCACHE	Allocate non-cached memory. This flag bit is similar in effect to calling MmAllocateContiguousMemorySpecifyCache with *CacheType* set to **MmNonCached**.
PAGE_WRITECOMBINE	Allocate write-combined memory. This flag bit is similar in effect to calling **MmAllocateContiguousMemorySpecifyCache** with *CacheType* set to **MmWriteCombined**.

If neither PAGE_NOCACHE nor PAGE_WRITECOMBINE is specified, the allocated memory is fully cached. In this case, the effect is similar to calling **MmAllocateContiguousMemorySpecifyCache** with *CacheType* set to **MmCached**.

`PreferredNode`

The preferred node number. If a multiprocessor system contains N nodes, the nodes are numbered from 0 to N-1. If the caller sets *PreferredNode* to MM_ANY_NODE_OK, the routine chooses which node to allocate memory from. Otherwise, if memory in the specified address range cannot be allocated from the preferred node, the routine returns **NULL**.

Return Value

MmAllocateContiguousNodeMemory returns the base virtual address for the allocated memory. If the request cannot be satisfied, the routine returns **NULL**.

Remarks

A kernel-mode device driver calls this routine to allocate a contiguous block of physical memory. The calling driver can specify whether to use no-execute (NX) memory for the allocation. In a non-uniform memory access (NUMA) multiprocessor system, the caller can specify a preferred node from which to allocate the memory. A node is a collection of processors that share fast access to a region of memory. In a non-NUMA multiprocessor or a single-processor system, **MmAllocateContiguousNodeMemory** treats all memory as belonging to a single node and allocates memory from this node.

MmAllocateContiguousNodeMemory allocates a block of nonpaged memory that is contiguous in physical address space. The routine maps this block to a contiguous block of virtual memory in the system address space and returns the virtual address of the base of this block. The routine aligns the starting address of a contiguous memory allocation to a memory page boundary.

Drivers must not access memory beyond the requested allocation size. For example, developers should not assume that their drivers can safely use memory between the end of their requested allocation and the next page boundary.

Because contiguous physical memory is usually in short supply, it should be used sparingly and only when necessary. A driver that must use contiguous memory should allocate this memory during driver initialization because physical memory is likely to become fragmented over time as the operating system allocates and frees memory. Typically, a driver calls **MmAllocateContiguousNodeMemory** from its DriverEntry routine to allocate an internal buffer for long-term use, and frees the buffer just before the driver is unloaded.

Memory allocated by **MmAllocateContiguousNodeMemory** must be freed when the memory is no longer needed. Call the MmFreeContiguousMemory routine to free memory that is allocated by **MmAllocateContiguousNodeMemory**.

MmAllocateContiguousNodeMemory is similar to the MmAllocateContiguousMemorySpecifyCacheNode routine. Unlike **MmAllocateContiguousMemorySpecifyCacheNode**, **MmAllocateContiguousNodeMemory** can be used to allocate no-execute (NX) memory. As a best practice, a driver should allocate NX memory unless the driver explicitly requires the ability to execute instructions in the allocated memory. By allocating NX memory, a driver improves security by preventing malicious software from executing instructions in this memory. Memory allocated by the MmAllocateContiguousMemory, MmAllocateContiguousMemorySpecifyCache, and **MmAllocateContiguousMemorySpecifyCacheNode** routines is always executable.

If you specify a nonzero value for the *BoundaryAddressMultiple* parameter, the physical address range of the allocated memory block will not cross an address boundary that is an integer multiple of this value. A driver should set this parameter to zero unless a nonzero value is required to work around a hardware limitation. For example, if a device cannot transfer data across 16-megabyte physical boundaries, the driver should specify a value of 0x1000000 for this parameter to ensure that the addresses that the device sees do not wrap around at a 16-megabyte boundary.

> **Note** Memory that **MmAllocateContiguousNodeMemory** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DriverEntry

MmAllocateContiguousMemory

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemory

MmAllocateMappingAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmAllocateMappingAddress** routine reserves a range of system virtual address space of the specified size.

Syntax

```
PVOID MmAllocateMappingAddress(
  SIZE_T NumberOfBytes,
  ULONG  PoolTag
);
```

Parameters

`NumberOfBytes`

Specifies the number of bytes to reserve.

`PoolTag`

Specifies a four-character tag used to identify the buffer. Use a distinct *PoolTag* tag for each allocation code path. For a description of pool tags, see ExAllocatePoolWithTag.

Return Value

MmAllocateMappingAddress returns a pointer to the beginning of the reserved memory buffer.

Remarks

MmAllocateMappingAddress reserves a system virtual address range for the caller to use. No physical memory is allocated for the virtual address range and the virtual memory cannot be accessed until it is mapped by the MmMapLockedPagesWithReservedMapping routine. The caller unmaps the reserved memory range by calling the MmUnmapReservedMapping routine. Finally, the caller can free the reserved range by calling MmFreeMappingAddress.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<=APC_LEVEL"

See Also

MmFreeMappingAddress

MmMapLockedPagesWithReservedMapping

MmUnmapReservedMapping

MmAllocateMdlForIoSpace function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocateMdlForIoSpace** routine allocates an MDL and initializes this MDL to describe a set of physical address ranges in I/O address space.

Syntax

```
NTSTATUS MmAllocateMdlForIoSpace(
  PMM_PHYSICAL_ADDRESS_LIST PhysicalAddressList,
  SIZE_T                    NumberOfEntries,
  PMDL                      *NewMdl
);
```

Parameters

`PhysicalAddressList`

A pointer to an array of MM_PHYSICAL_ADDRESS_LIST structures that describe the physical address ranges to include in the allocated MDL.

`NumberOfEntries`

The number of elements in the **MM_PHYSICAL_ADDRESS_LIST** array pointed to by *PhysicalAddressList*.

`NewMdl`

A pointer to a location to which the routine writes a pointer to the newly allocated MDL.

Return Value

MmAllocateMdlForIoSpace returns STATUS_SUCCESS if it is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER_1	A physical address is not aligned to a page boundary; or a physical address range is not a multiple of the page size; or a physical address range is used by the operating system for RAM and is not available for use as I/O space.

STA TUS _IN SUF FICI ENT _RE SO URC ES	Insufficient system resources are available to perform the requested operation.

Do not assume that the preceding list of error return codes is exhaustive. The routine might return error codes that do not appear in the list.

Remarks

This routine accepts, as an input parameter, an array of **MM_PHYSICAL_ADDRESS_LIST** structures that describe a set of physical address ranges in I/O address space, and allocates an MDL that describes these ranges. Consecutive physical address ranges in the array are not required to be contiguous.

The physical address ranges in the *PhysicalAddressList* array must satisfy the following conditions:

- The base physical address for each range must be aligned to a PAGE_SIZE boundary in memory.
- The size, in bytes, of each range must be an integer multiple of PAGE_SIZE.
- All physical address ranges must be in memory that is available for use as I/O address space. They cannot be in memory space that is used by the operating system for RAM.
- The total size of all the ranges must be less than 4 gigabytes. Specifically, the total size must not exceed 2^{32}-1 bytes.

The caller is responsible for freeing the allocated MDL when it is no longer needed. To free the MDL, call the IoFreeMdl routine. For more information about MDLs, see Using MDLs.

The MDL that is created by **MmAllocateMdlForIoSpace** is not mapped to virtual memory, but can be supplied to a routine such as MapTransferEx to initiate a DMA transfer to or from the physical memory ranges described by the MDL. To map this MDL to a contiguous range of virtual addresses so that it can be accessed by the processor, call the MmMapLockedPagesSpecifyCache routine.

Only ranges of the physical address space that are not reserved by the operating system for use as memory are available to drivers for use as I/O address space. Drivers use I/O address space to access memory-mapped hardware resources such as device registers. When a driver starts, it might receive one or more physical address ranges as translated hardware resources. For more information, see Mapping Bus-Relative Addresses to Virtual Addresses.

In some processor architectures, such as the x86, devices can be either memory-mapped or mapped to port addresses in a special I/O address space that is dedicated to devices and is separate from the memory address space. Drivers can use **MmAllocateMdlForIoSpace** to allocate MDLs only for memory-mapped devices.

Examples

The following code example shows how to construct an array of MM_PHYSICAL_ADDRESS_LIST structures that describe the physical address ranges to include in the allocated MDL.

```
extern ULONG64 BasePhysicalAddress;
extern SIZE_T ChunkSize;
extern SIZE_T Stride;

#define ARRAYSIZE(x)  (sizeof(x)/sizeof((x)[0]))

NTSTATUS Status;
PMDL Mdl;
MM_PHYSICAL_ADDRESS_LIST AddressList[3];

AddressList[0].PhysicalAddress.QuadPart = BasePhysicalAddress;
AddressList[0].NumberOfBytes = ChunkSize;

BasePhysicalAddress += Stride;

AddressList[1].PhysicalAddress.QuadPart = BasePhysicalAddress;
AddressList[1].NumberOfBytes = ChunkSize;

BasePhysicalAddress += Stride;

AddressList[2].PhysicalAddress.QuadPart = BasePhysicalAddress;
AddressList[2].NumberOfBytes = ChunkSize;

Status = MmAllocateMdlForIoSpace (AddressList, ARRAYSIZE(AddressList), &Mdl);
```

In this example, the starting physical address is specified by the `BasePhysicalAddress` variable. The number of bytes in each physical address range is specified by the `ChunkSize` variable. The byte offset from the start of one physical range to the start of the next is specified by the `Stride` variable. `BasePhysicalAddress` must be aligned to a page boundary in memory, and `ChunkSize` and `Stride` must be multiples of the page size.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

IoFreeMdl

MDL

MM_PHYSICAL_ADDRESS_LIST

MapTransferEx

MmMapLockedPagesSpecifyCache

MmAllocateNodePagesForMdlEx function

1/25/2019 • 4 minutes to read • Edit Online

The **MmAllocateNodePagesForMdlEx** routine allocates nonpaged physical memory from an ideal node, and allocates an MDL structure to describe this memory.

Syntax

```
PMDL MmAllocateNodePagesForMdlEx(
  PHYSICAL_ADDRESS    LowAddress,
  PHYSICAL_ADDRESS    HighAddress,
  PHYSICAL_ADDRESS    SkipBytes,
  SIZE_T              TotalBytes,
  MEMORY_CACHING_TYPE CacheType,
  ULONG               IdealNode,
  ULONG               Flags
);
```

Parameters

`LowAddress`

The physical address of the start of the first address range from which the allocated pages can come. If **MmAllocateNodePagesForMdlEx** cannot allocate the requested number of bytes in the first address range, the routine iterates through additional address ranges to get more pages. At each iteration, **MmAllocateNodePagesForMdlEx** adds the value of *SkipBytes* to the previous start address to calculate the start of the next address range.

`HighAddress`

The physical address of the end of the first address range that the allocated pages can come from.

`SkipBytes`

The number of bytes to skip from the start of the previous address range that the allocated pages can come from. *SkipBytes* must be an integer multiple of the virtual memory page size, in bytes.

`TotalBytes`

The total number of bytes to allocate for the MDL.

`CacheType`

A MEMORY_CACHING_TYPE value, which indicates the type of caching that is allowed for the requested memory.

`IdealNode`

The ideal node number. If a multiprocessor system contains N nodes, valid node numbers are in the range 0 to N-1. Your driver can call the KeQueryHighestNodeNumber routine to get the highest node number. A single-processor or non-NUMA multiprocessor system has only one node, node 0, from which to allocate memory. For a NUMA multiprocessor system, the allocation is made from the ideal node, if possible. If insufficient memory is available in the ideal node to satisfy the allocation request, and the caller does not set the MM_ALLOCATE_FROM_LOCAL_NODE_ONLY flag, **MmAllocateNodePagesForMdlEx** will try to allocate memory from other nodes.

`Flags`

Flags for this operation. Set this parameter to zero or to the bitwise-OR of one or more of the following flag bits:

- MM_DONT_ZERO_ALLOCATION
- MM_ALLOCATE_FROM_LOCAL_NODE_ONLY
- MM_ALLOCATE_FULLY_REQUIRED
- MM_ALLOCATE_NO_WAIT
- MM_ALLOCATE_PREFER_CONTIGUOUS
- MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS

The MM_ALLOCATE_PREFER_CONTIGUOUS and MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS flag bits are mutually exclusive. For more information about these flags, see MM_ALLOCATE_XXX.

Return Value

MmAllocateNodePagesForMdlEx returns a pointer to an MDL structure if it is successful. Otherwise, if the routine fails to allocate any memory, the routine returns **NULL**.

A return value of **NULL** indicates that no physical memory pages are available in the specified address ranges, or that there is not enough memory pool available to allocate the MDL structure.

If the routine successfully allocates some, but not all, of the requested memory, the MDL describes as much physical memory as the routine was able to allocate.

Remarks

In a non-uniform memory access (NUMA) multiprocessor system, the caller can specify an ideal node from which to allocate the memory. A node is a collection of processors that share fast access to a region of memory. In a non-NUMA multiprocessor or a single-processor system, **MmAllocateNodePagesForMdlEx** treats all memory as belonging to a single node and allocates memory from this node.

By default, the physical memory pages that **MmAllocateNodePagesForMdlEx** returns are not contiguous pages. Callers can override the default behavior of this routine by setting the MM_ALLOCATE_PREFER_CONTIGUOUS or MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS flag bit in the *Flags* parameter.

MmAllocateNodePagesForMdlEx does not map the allocated physical memory into virtual memory. If necessary, the caller can call a routine such as MmMapLockedPagesSpecifyCache to map the physical memory pages described by the MDL.

MmAllocateNodePagesForMdlEx is designed for kernel-mode drivers that do not need corresponding virtual addresses (that is, they need physical pages and do not need them to be physically contiguous), and for kernel-mode drivers that can achieve substantial performance gains if physical memory for a device is allocated in a specific physical address range (for example, an AGP graphics card).

Depending on how much physical memory is currently available in the requested ranges, **MmAllocateNodePagesForMdlEx** might return an MDL that describes less memory than was requested. The routine also might return **NULL** if no memory was allocated. The caller should check the amount of memory that is actually allocated, as described by the MDL.

The caller must use MmFreePagesFromMdl to release the memory pages that are described by an MDL that was created by **MmAllocateNodePagesForMdlEx**. After calling **MmFreePagesFromMdl**, the caller must also call ExFreePool to release the memory allocated for the MDL structure.

By default, **MmAllocateNodePagesForMdlEx** fills the pages that it allocates with zeros. The caller can specify the MM_DONT_ZERO_ALLOCATION flag to override this default and to possibly improve performance.

> **Note** Memory that **MmAllocateNodePagesForMdlEx** allocates is uninitialized if you specify the MM_DONT_ZERO_ALLOCATION flag. A kernel-mode driver must first zero this memory if the driver is going to make the memory visible to user-mode software (to avoid leaking potentially privileged contents). For more information about this flag, see MM_ALLOCATE_XXX.

The maximum amount of memory that **MmAllocateNodePagesForMdlEx** can allocate in a single call is (4 gigabytes - PAGE_SIZE). The routine can satisfy an allocation request for this amount only if enough pages are available.

MmAllocateNodePagesForMdlEx runs at IRQL <= APC_LEVEL. If necessary, your driver can call **MmAllocateNodePagesForMdlEx** at DISPATCH_LEVEL. However, you can improve driver performance by calling at APC_LEVEL or below.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL (See Remarks section.)"

See Also

ExFreePool

KeQueryHighestNodeNumber

MDL

MEMORY_CACHING_TYPE

MM_ALLOCATE_XXX

MmFreePagesFromMdl

MmMapLockedPagesSpecifyCache

MmAllocatePagesForMdl function

1/25/2019 • 3 minutes to read • Edit Online

The **MmAllocatePagesForMdl** routine allocates zero-filled, nonpaged, physical memory pages to an MDL.

Syntax

```
PMDL MmAllocatePagesForMdl(
  PHYSICAL_ADDRESS LowAddress,
  PHYSICAL_ADDRESS HighAddress,
  PHYSICAL_ADDRESS SkipBytes,
  SIZE_T           TotalBytes
);
```

Parameters

`LowAddress`

Specifies the physical address of the start of the first address range from which the allocated pages can come. If **MmAllocatePagesForMdl** cannot allocate the requested number of bytes in the first address range, it iterates through additional address ranges to get more pages. At each iteration, **MmAllocatePagesForMdl** adds the value of *SkipBytes* to the previous start address to obtain the start of the next address range.

`HighAddress`

Specifies the physical address of the end of the first address range from which the allocated pages can come.

`SkipBytes`

Specifies the number of bytes to skip from the start of the previous address range from which the allocated pages can come. *SkipBytes* must be an integer multiple of the virtual memory page size, in bytes.

`TotalBytes`

Specifies the total number of bytes to allocate for the MDL.

Return Value

MmAllocatePagesForMdl returns one of the following:

RETURN CODE	DESCRIPTION
MDL pointer	The MDL pointer describes a set of physical pages in the specified address range. If the requested number of bytes is not available, the MDL describes as much physical memory as is available.
NULL	There are no physical memory pages in the specified address ranges, or there is not enough memory pool for the MDL itself.

Remarks

Drivers that are running in Windows Server 2003 Service Pack 1 (SP1) and later versions of Windows should use the MmAllocatePagesForMdlEx routine instead of **MmAllocatePagesForMdl**. **MmAllocatePagesForMdlEx** provides better performance than **MmAllocatePagesForMdl** by avoiding unnecessary flushes of the TLB and cache memory.

The physical memory pages that are returned by **MmAllocatePagesForMdl** are typically not contiguous pages. **MmAllocatePagesForMdl** always fills the allocated pages in the returned MDL with zeros.

MmAllocatePagesForMdl is designed to be used by kernel-mode drivers that do not need corresponding virtual addresses (that is, they need physical pages and do not need the pages to be physically contiguous) or by kernel-mode drivers that can achieve substantial performance gains if physical memory for a device is allocated in a specific physical address range. A driver for an AGP graphics card is an example of such a driver.

Depending on how much physical memory is currently available in the requested ranges, **MmAllocatePagesForMdl** might return an MDL that describes less memory than was requested. The routine returns **NULL** if no memory was allocated. The caller should check the amount of memory that is actually allocated to the MDL.

The caller must use MmFreePagesFromMdl to release the memory pages that are described by an MDL that was created by **MmAllocatePagesForMdl**. After calling **MmFreePagesFromMdl**, the caller must also call ExFreePool to release the memory that is allocated for the MDL structure itself.

In Windows 2000 and later versions of Windows, the maximum amount of memory that **MmAllocatePagesForMdl** can allocate in a single call is (4 gigabytes - PAGE_SIZE). The routine can satisfy an allocation request for this amount only if enough pages are available.

MmAllocatePagesForMdl runs at IRQL <= APC_LEVEL. Windows Server 2008 and later versions of the Windows operating system enable **MmAllocatePagesForMdl** callers to call at DISPATCH_LEVEL. However, you can improve driver performance by calling at APC_LEVEL or below.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows; however, MmAllocatePagesForMdlEx should be used instead of MmAllocatePagesForMdl in Windows Server 2003 Service Pack 1 (SP1) and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs, SpNoWait, StorPortStartIo

See Also

ExFreePool

MmAllocatePagesForMdlEx

MmFreePagesFromMdl

MmMapLockedPages

MmAllocatePagesForMdlEx function

1/25/2019 • 6 minutes to read • Edit Online

The **MmAllocatePagesForMdlEx** routine allocates nonpaged, physical memory pages to an MDL.

Syntax

```
PMDL MmAllocatePagesForMdlEx(
  PHYSICAL_ADDRESS    LowAddress,
  PHYSICAL_ADDRESS    HighAddress,
  PHYSICAL_ADDRESS    SkipBytes,
  SIZE_T              TotalBytes,
  MEMORY_CACHING_TYPE CacheType,
  ULONG               Flags
);
```

Parameters

`LowAddress`

Specifies the physical address of the start of the first address range from which the allocated pages can come. If **MmAllocatePagesForMdlEx** cannot allocate the requested number of bytes in the first address range, it iterates through additional address ranges to get more pages. At each iteration, **MmAllocatePagesForMdlEx** adds the value of *SkipBytes* to the previous start address to obtain the start of the next address range.

`HighAddress`

Specifies the physical address of the end of the first address range that the allocated pages can come from.

`SkipBytes`

Specifies the number of bytes to skip from the start of the previous address range that the allocated pages can come from. *SkipBytes* must be an integer multiple of the virtual memory page size, in bytes.

`TotalBytes`

Specifies the total number of bytes to allocate for the MDL.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching that is allowed for the requested memory.

`Flags`

Specifies flags for this operation. Set this parameter to zero or to the bitwise OR of one or more of the following **MM_ALLOCATE_*XXX*** flag bits:

The last four items in the preceding list are supported only in Windows 7 and later versions of Windows.

VALUE	MEANING

VALUE	MEANING
MM_DONT_ZERO_ALLOCATION 0x00000001	Do not fill the allocated pages with zeros. By default, MmAllocatePagesForMdlEx zeros the pages that it allocates. By skipping this operation, you can potentially improve the performance of the MmAllocatePagesForMdlEx call. However, you must not use this flag unless either you never expose the allocated pages to user-mode programs, or you always overwrite the original contents of the pages before you expose the allocated pages to user-mode programs.
MM_ALLOCATE_FROM_LOCAL_NODE_ONLY 0x00000002	Allocate pages only from the ideal node. This flag applies only to multiprocessor systems that have non-uniform memory access (NUMA) architectures. Starting with Windows Vista, this flag indicates that all pages must be allocated from the ideal node of the current thread. No pages are to be allocated from other nodes. In versions of Windows earlier than Windows Vista, this flag indicates that all pages must be allocated from the local node; that is, from the node that the current processor belongs to. For more information about NUMA multiprocessor systems, see NUMA Support .
MM_ALLOCATE_FULLY_REQUIRED 0x00000004	A full allocation is required. Starting with Windows 7, this flag requires MmAllocatePagesForMdlEx to return NULL if it cannot allocate all the requested pages. The routine returns a non-NULL value only if it successfully obtains the entire requested allocation. This flag enables the memory manager to perform the allocation more efficiently in cases in which the caller requires a full allocation.
MM_ALLOCATE_NO_WAIT 0x00000008	Do not wait. Starting with Windows 7, this flag indicates that the MmAllocatePagesForMdlEx call must not block the calling thread. Typically, the caller is a kernel-mode driver that is running at IRQL < DISPATCH_LEVEL but cannot allow its execution to be blocked. For example, the driver might be assisting with paging or power-management operations. Regardless of whether this flag is set, MmAllocatePagesForMdlEx never blocks callers that are running at IRQL = DISPATCH_LEVEL.
MM_ALLOCATE_PREFER_CONTIGUOUS 0x00000010	Allocation is performed in a way that minimizes system memory fragmentation. Starting with Windows 7, this flag indicates that the caller wants to avoid fragmenting physical memory to make more contiguous memory available to other callers. The allocated pages are not guaranteed to be (and usually are not) physically contiguous, even if plenty of contiguous memory is available. Callers that require contiguous memory should specify MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS instead of MM_ALLOCATE_PREFER_CONTIGUOUS.

VALUE	MEANING
MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS 0x00000020	Contiguous memory is required. Starting with Windows 7, this flag indicates that the requested pages must be allocated as contiguous blocks of physical memory. If the SkipBytes parameter is zero, MmAllocatePagesForMdlEx either succeeds and returns a single, contiguous block, or it fails and returns NULL. It never returns a partial allocation. For SkipBytes = 0, the allocated pages satisfy the address range requirements that are specified by the LowAddress and HighAddress parameters, but the pages are subject to no special alignment restrictions. If SkipBytes is nonzero, SkipBytes must be a power of two and must be greater than or equal to PAGE_SIZE, and the TotalBytes parameter value must be a multiple of SkipBytes. In this case, the returned MDL can contain multiple blocks of contiguous pages. That is, each block is internally contiguous but the blocks are not necessarily contiguous with each other. Each block of contiguous pages is guaranteed to be exactly SkipBytes long and to be aligned on a SkipBytes boundary. Partial allocations can occur if SkipBytes is nonzero, but each contiguous block in a partial allocation is guaranteed to be SkipBytes long.
MM_ALLOCATE_FAST_LARGE_PAGES 0x00000040	Starting with Windows 8, this flag specifies that the allocation must be satisfied from the operating system's large page cache. If the cache is empty, allocation fails. If MM_ALLOCATE_FAST_LARGE_PAGES is not specified, **MmAllocatePagesForMdlEx** uses cached large pages if they are available. If the cache is exhausted, **MmAllocatePagesForMdlEx** attempts to construct additional large pages, which may take a long time. MM_ALLOCATE_FAST_LARGE_PAGES must be used with the MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS flag. The *SkipBytes* parameter must be set to a multiple of large page size.

Return Value

MmAllocatePagesForMdlEx returns one of the following:

RETURN CODE	DESCRIPTION
MDL pointer	A non-**NULL** return value is a pointer to an MDL that describes a set of physical pages in the specified address range. If the requested number of bytes is not available, the MDL describes as much physical memory as is available.
NULL	Indicates that no physical memory pages are available in the specified address ranges, or that there is not enough memory pool for the MDL itself.

Remarks

By default, the physical memory pages that **MmAllocatePagesForMdlEx** returns are not contiguous pages. Starting with Windows 7, callers can override the default behavior of this routine by setting the MM_ALLOCATE_PREFER_CONTIGUOUS or MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS flag bit in the *Flags* parameter.

MmAllocatePagesForMdlEx is designed for kernel-mode drivers that do not need corresponding virtual addresses (that is, they need physical pages and do not need them to be physically contiguous), and for kernel-mode drivers that can achieve substantial performance gains if physical memory for a device is allocated in a specific physical address range (for example, an AGP graphics card).

Depending on how much physical memory is currently available in the requested ranges, **MmAllocatePagesForMdlEx** might return an MDL that describes less memory than was requested. The routine also might return **NULL** if no memory was allocated. The caller should check the amount of memory that is actually allocated to the MDL.

The caller must use MmFreePagesFromMdl to release the memory pages that are described by an MDL that was created by **MmAllocatePagesForMdlEx**. After calling **MmFreePagesFromMdl**, the caller must also call ExFreePool to release the memory that is allocated for the MDL structure.

By default, **MmAllocatePagesForMdlEx** fills the pages that it allocates with zeros. The caller can specify the MM_DONT_ZERO_ALLOCATION flag to override this default and to possibly improve performance.

> **Note** Memory that **MmAllocatePagesForMdlEx** allocates is uninitialized if you specify the MM_DONT_ZERO_ALLOCATION flag. A kernel-mode driver must first zero this memory if the driver is going to make the memory visible to user-mode software (to avoid leaking potentially privileged contents). For more information about this flag, see MM_ALLOCATE_XXX.

The maximum amount of memory that **MmAllocatePagesForMdlEx** can allocate in a single call is (4 gigabytes - PAGE_SIZE). The routine can satisfy an allocation request for this amount only if enough pages are available.

MmAllocatePagesForMdlEx runs at IRQL <= APC_LEVEL. In Windows Server 2008 and later versions of Windows, callers of **MmAllocatePagesForMdlEx** are allowed to be at DISPATCH_LEVEL. However, you can improve driver performance by calling at APC_LEVEL or below.

Requirements

Minimum supported client	Available in Windows Server 2003 with Service Pack 1 (SP1) and later versions of Windows. You should use this routine instead of MmAllocatePagesForMdl on these operating systems.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlMmApcLte

See Also

ExFreePool

MEMORY_CACHING_TYPE

MmAllocatePagesForMdl

MmFreePagesFromMdl

MmMapLockedPages

MmBuildMdlForNonPagedPool function

1/25/2019 • 2 minutes to read • Edit Online

The **MmBuildMdlForNonPagedPool** routine receives an MDL that specifies a nonpaged virtual memory buffer, and updates it to describe the underlying physical pages.

Syntax

```
void MmBuildMdlForNonPagedPool(
  PMDL MemoryDescriptorList
);
```

Parameters

`MemoryDescriptorList`

A pointer to an MDL that specifies a virtual memory buffer in nonpaged memory. The caller used the IoAllocateMdl routine to create the MDL for this buffer.

Return Value

None

Remarks

At entry, the specified MDL must describe a buffer in nonpaged system memory or in memory that is locked down (the physical pages cannot be traded), such as memory that is allocated by the ExAllocatePoolWithTag routine with *PoolType* = **NonPagedPool** or by the MmAllocateContiguousMemorySpecifyCache routine. **MmBuildMdlForNonPagedPool** updates the MDL to describe the underlying physical pages.

MmBuildMdlForNonPagedPool may not be used with MDLs describing buffers allocated on a kernel stack. To build an MDL describing a kernel stack buffer, drivers must call MmProbeAndLockPages. This is because kernel stack pages can be traded unless they are probe-and-locked. This rule applies even if the driver guarantees that the kernel stack cannot be paged out.

Because the pages described by the MDL are already nonpageable and are already mapped to the system address space, drivers must not try to lock them by using the MmProbeAndLockPages routine, or to create additional system-address-space mappings by using the MmMapLockedPagesSpecifyCache routine. Similarly, drivers must not try to unlock the pages by using the MmUnlockPages routine, or to release the existing system-address-space mapping by using the MmUnmapLockedPages routine. If a driver performs any of these illegal operations on an MDL that is built by **MmBuildMdlForNonPagedPool,** the resulting behavior is undefined.

Passing an MDL built by **MmBuildMdlForNonPagedPool** to the MmGetSystemAddressForMdlSafe routine is allowed. The **MmGetSystemAddressForMdlSafe** call, in this case, simply returns the starting virtual address of the buffer that is described by the MDL.

A driver can use the **MmMapLockedPagesSpecifyCache** routine to map an MDL that is built by **MmBuildMdlForNonPagedPool** into user virtual address space. However, the driver must perform this operation in a way that avoids certain security issues. For more information, see MmMapLockedPagesSpecifyCache.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

ExAllocatePoolWithTag

MmAllocateContiguousMemorySpecifyCache

MmGetSystemAddressForMdlSafe

MmMapLockedPagesSpecifyCache

MmProbeAndLockPages

MmUnlockPages

MmUnmapLockedPages

MmFreeContiguousMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeContiguousMemory** routine releases a range of physically contiguous memory that was allocated by an **MmAllocateContiguousMemory*Xxx*** routine.

Syntax

```
void MmFreeContiguousMemory(
  PVOID BaseAddress
);
```

Parameters

`BaseAddress`

Pointer to the virtual address of the memory to be freed.

Return Value

None

Remarks

The **MmFreeContiguousMemory** routine frees a block of physically contiguous memory that was allocated by a previous call to the MmAllocateContiguousMemory, MmAllocateContiguousMemorySpecifyCache, or MmAllocateContiguousMemorySpecifyCacheNode routine. The *BaseAddress* parameter must be the base address that was obtained from the previous call to the **MmAllocateContiguousMemory*Xxx*** routine.

A device driver that must use contiguous memory should allocate only what it needs during driver initialization because physical memory is likely to become fragmented as the system runs. Such a driver must deallocate the memory when the driver is done using the memory.

Callers of **MmFreeContiguousMemory** must be running at IRQL = APC_LEVEL. For Windows Server 2008 and later versions of the Windows operating system, you can also call **MmFreeContiguousMemory** with IRQL <= DISPATCH_LEVEL. However, you can improve driver performance by calling at APC_LEVEL or below.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	IrqlMmDispatch, HwStorPortProhibitedDDIs

See Also

MmAllocateContiguousMemory

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemorySpecifyCache function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeContiguousMemorySpecifyCache** routine frees a buffer that was allocated by an **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

Syntax

```
void MmFreeContiguousMemorySpecifyCache(
  PVOID               BaseAddress,
  SIZE_T              NumberOfBytes,
  MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`BaseAddress`

Specifies the base address of the buffer to be freed. Must match the address returned by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** call that allocated the buffer.

`NumberOfBytes`

Specifies the size in bytes of the buffer to be freed. Must match the size requested when the buffer was allocated by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

`CacheType`

Specifies the cache type of the buffer to be freed. Must match the cache type requested when the buffer was allocated by the **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine.

Return Value

None

Remarks

The **MmFreeContiguousMemorySpecifyCache** routine frees a block of physically contiguous memory that was allocated by a previous call to the MmAllocateContiguousMemorySpecifyCache or MmAllocateContiguousMemorySpecifyCacheNode routine. However, MmFreeContiguousMemory is the preferred routine to use to free memory that was allocated by an **MmAllocateContiguousMemorySpecifyCache*Xxx*** routine. **MmFreeContiguousMemory** is faster than **MmFreeContiguousMemorySpecifyCache** and requires fewer parameters.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemorySpecifyCache

MmAllocateContiguousMemorySpecifyCacheNode

MmFreeContiguousMemory

MmFreeMappingAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreeMappingAddress** routine frees a range of virtual memory reserved by the MmAllocateMappingAddress routine.

Syntax

```
void MmFreeMappingAddress(
  PVOID BaseAddress,
  ULONG PoolTag
);
```

Parameters

`BaseAddress`

Pointer to the beginning of the reserved memory buffer to free. This must be an address previously returned by MmAllocateMappingAddress.

`PoolTag`

Specifies the pool tag for the reserved memory buffer. This must be identical to the value specified in the *PoolTag* parameter of the call to MmAllocateMappingAddress that reserved the buffer.

Return Value

None

Remarks

MmFreeMappingAddress frees a range of memory reserved by MmAllocateMappingAddress. If the memory range has already been mapped by MmMapLockedPagesWithReservedMapping, it must first be unmapped with MmUnmapReservedMapping before the memory range can be freed.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<=APC_LEVEL"

See Also

MmAllocateMappingAddress

MmMapLockedPagesWithReservedMapping

MmUnmapReservedMapping

MmFreePagesFromMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **MmFreePagesFromMdl** routine frees all the physical pages that are described by an MDL that was created by the MmAllocatePagesForMdl routine.

Syntax

```
void MmFreePagesFromMdl(
  PMDL MemoryDescriptorList
);
```

Parameters

`MemoryDescriptorList`

Pointer to an MDL that was created by **MmAllocatePagesForMdl**.

Return Value

None

Remarks

MmFreePagesFromMdl can only be used to free the memory pages that are described by an MDL that was created by **MmAllocatePagesForMdl**.

After calling **MmFreePagesFromMdl**, the caller must also call ExFreePool to release the memory that was allocated for the MDL structure.

MmFreePagesFromMdl runs at IRQL <= APC_LEVEL. For Windows Server 2008 and later versions of the Windows operating system, you can also call this routine at DISPATCH_LEVEL. However, you can improve driver performance by calling at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

ExFreePool

MmAllocatePagesForMdl

MmGetSystemAddressForMdl macro

1/25/2019 • 2 minutes to read • Edit Online

The **MmGetSystemAddressForMdl** routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me. It is supported only for WDM drivers that must run on Windows 98. Otherwise, use MmGetSystemAddressForMdlSafe.

MmGetSystemAddressForMdl is a macro that returns a nonpaged system-space virtual address for the buffer described by the MDL. It maps the physical pages described by a given MDL into system space, if they are not already mapped to system space.

Syntax

```
void MmGetSystemAddressForMdl(
   MDL
);
```

Parameters

`MDL`

Pointer to a buffer whose corresponding base virtual address is to be mapped.

Return Value

MmGetSystemAddressForMdl returns the base system-space virtual address that maps the physical pages described by the given MDL.

Remarks

Drivers of PIO devices call this routine to translate a virtual address range, described by the MDL at **Irp->MdlAddress**, for a user buffer to a system-space address range.

The MDL must describe nonpageable memory. In other words, the input MDL must describe an already locked-down user-space buffer returned by MmProbeAndLockPages, a locked-down buffer returned by MmBuildMdlForNonPagedPool, or system-space memory allocated from nonpaged pool, contiguous memory, or noncached memory.

The returned base address has the same offset as the virtual address in the MDL.

Windows 2000 issues a bug check if the attempt to map to system space fails. (Therefore, you should use MmGetSystemAddressForMdlSafe instead). In Windows 98, this routine returns **NULL** in case of failure.

Requirements

Minimum supported client	Obsolete for Windows 2000 and later versions of Windows, and for Windows Me. This routine is supported only for WDM drivers that must run on Windows 98. Otherwise, use MmGetSystemAddressForMdlSafe.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<=DISPATCH_LEVEL"

See Also

MmBuildMdlForNonPagedPool

MmGetSystemAddressForMdlSafe

MmProbeAndLockPages

MmGetSystemRoutineAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmGetSystemRoutineAddress** routine returns a pointer to a function specified by *SystemRoutineName*.

Syntax

```
PVOID MmGetSystemRoutineAddress(
  PUNICODE_STRING SystemRoutineName
);
```

Parameters

`SystemRoutineName`

Specifies the name of the system routine to resolve.

Return Value

If the function name can be resolved, the routine returns a pointer to the function. Otherwise, the routine returns **NULL**.

Remarks

Drivers can use this routine to determine if a routine is available on a specific version of Windows. It can only be used for routines exported by the kernel or HAL, not for any driver-defined routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

IoIsWdmVersionAvailable

MmIsDriverSuspectForVerifier function

1/25/2019 • 2 minutes to read • Edit Online

The **MmIsDriverSuspectForVerifier** routine indicates whether the driver represented by the specified driver object is in the list of drivers that are selected to be verified by Driver Verifier.

Syntax

```
LOGICAL MmIsDriverSuspectForVerifier(
  _DRIVER_OBJECT *DriverObject
);
```

Parameters

`DriverObject`

A pointer to the driver object. This object is a DRIVER_OBJECT structure that represents a loaded driver image. The driver receives this pointer as an input parameter to its DriverEntry routine.

Return Value

MmIsDriverSuspectForVerifier returns **TRUE** if the driver is selected for verification. Otherwise, the routine returns **FALSE**.

Remarks

A kernel-mode driver can call this routine to determine whether it is being monitored by Driver Verifier. Driver Verifier monitors kernel-mode drivers to detect illegal function calls or actions that might corrupt the system. To select drivers to be verified, you can use the Verifier Command Line or Driver Verifier Manager. For more information about adding drivers to the driver verification list, see Selecting Drivers to be Verified.

A related routine, MmIsDriverVerifying, returns **TRUE** if the specified driver either is in the driver verification list or imports calls to entry points in a driver that is in the driver verification list. Otherwise, **MmIsDriverVerifying** returns **FALSE**.

For example, if driver A has an import table through which it calls one or more entry points in driver B, and driver B is in the driver verification list, then `MmIsDriverVerifying(A)` returns **TRUE** and `MmIsDriverSuspectForVerifier(B)` returns **TRUE**. If driver A is not in the driver verification list, `MmIsDriverSuspectForVerifier(A)` returns **FALSE**. Even if driver B does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifying(B)` returns **TRUE** because driver B is in the driver verification list. If a driver C is not in the driver verification list and does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifying(C)` and `MmIsDriverSuspectForVerifier(C)` both return **FALSE**.

Another related routine, MmIsDriverVerifyingByAddress, indicates whether a driver identified by a driver image address is being verified or calls a driver that is being verified. **MmIsDriverVerifyingByAddress** is available starting with Windows Vista.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DRIVER_OBJECT

DriverEntry

MmIsDriverVerifying

MmIsDriverVerifyingByAddress

MmIsDriverVerifying function

1/25/2019 • 2 minutes to read • Edit Online

The **MmIsDriverVerifying** routine indicates whether the kernel-mode driver that is identified by the specified driver object is being verified or calls a driver that is being verified by Driver Verifier.

Syntax

```
LOGICAL MmIsDriverVerifying(
  _DRIVER_OBJECT *DriverObject
);
```

Parameters

`DriverObject`

A pointer to a DRIVER_OBJECT structure that represents a driver object. The driver receives this pointer as an input parameter to its DriverEntry routine.

Return Value

MmIsDriverVerifying returns **TRUE** if the specified driver either is in the driver verification list or imports calls to entry points in a driver that is in the driver verification list. Otherwise, this routine returns **FALSE**.

Remarks

A kernel-mode driver can call this routine to determine whether it is being monitored by Driver Verifier. Driver Verifier monitors kernel-mode drivers to detect illegal function calls or actions that might corrupt the system. To select drivers to be verified, you can use the Verifier Command Line or Driver Verifier Manager. For more information about adding drivers to the driver verification list, see Selecting Drivers to be Verified.

A similar routine, MmIsDriverVerifyingByAddress, indicates whether a driver identified by a driver image address is being verified or calls a driver that is being verified. **MmIsDriverVerifyingByAddress** is available starting with Windows Vista.

Another related routine, MmIsDriverSuspectForVerifier, indicates whether a driver represented by a driver object is in the list of drivers that are selected to be verified. **MmIsDriverSuspectForVerifier** is available starting with Windows 8.

For example, if driver A has an import table through which it calls one or more entry points in driver B, and driver B is in the driver verification list, then `MmIsDriverVerifying(A)` returns **TRUE** and `MmIsDriverSuspectForVerifier(B)` returns **TRUE**. If driver A is not in the driver verification list, `MmIsDriverSuspectForVerifier(A)` returns **FALSE**. Even if driver B does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifying(B)` returns **TRUE** because driver B is in the driver verification list. If a driver C is not in the driver verification list and does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifying(C)` and `MmIsDriverSuspectForVerifier(C)` both return **FALSE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

DRIVER_OBJECT

DriverEntry

MmIsDriverVerifyingByAddress

MmIsDriverVerifyingByAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmIsDriverVerifyingByAddress** routine checks whether the kernel-mode driver that is identified by the specified image address is being verified or calls a driver that is being verified by Driver Verifier.

Syntax

```
LOGICAL MmIsDriverVerifyingByAddress(
  PVOID AddressWithinSection
);
```

Parameters

`AddressWithinSection`

A pointer to the virtual address within the driver image. **MmIsDriverVerifyingByAddress** uses this address to determine which driver to check.

Return Value

MmIsDriverVerifyingByAddress returns **TRUE** if the specified driver either is in the driver verification list or imports calls to entry points in a driver that is in the driver verification list. Otherwise, this routine returns **FALSE**.

Remarks

A kernel-mode driver can call this routine to determine whether it or another driver is being monitored by Driver Verifier. Driver Verifier monitors kernel-mode drivers to detect illegal function calls or actions that might corrupt the system. To select drivers to be verified, you can use the Verifier Command Line or Driver Verifier Manager. For more information about adding drivers to the driver verification list, see Selecting Drivers to be Verified.

A similar routine, MmIsDriverVerifying, indicates whether a driver identified by a driver object is being verified or calls a driver that is being verified.

Another related routine, MmIsDriverSuspectForVerifier, indicates whether a driver represented by a driver object is in the list of drivers that are selected to be verified.

For example, if driver A has an import table through which it calls one or more entry points in driver B, and driver B is in the driver verification list, then `MmIsDriverVerifyingByAddress(Aobj)` returns **TRUE** and `MmIsDriverSuspectForVerifier(Badr)` returns **TRUE**, where `Aobj` is a pointer to the driver object for A and `Badr` is an address in driver B. If driver A is not in the driver verification list, `MmIsDriverSuspectForVerifier(Aobj)` returns **FALSE**. Even if driver B does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifyingByAddress(Badr)` returns **TRUE** because driver B is in the driver verification list. If a driver C is not in the driver verification list and does not call entry points in any drivers that are in the driver verification list, `MmIsDriverVerifyingByAddress(Cadr)` and `MmIsDriverSuspectForVerifier(Cobj)` both return **FALSE**.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

MmIsDriverSuspectForVerifier

MmIsDriverVerifying

MmLockPagableCodeSection macro

1/25/2019 • 5 minutes to read • Edit Online

The **MmLockPagableCodeSection** routine locks a section of driver code, containing a set of driver routines marked with a special compiler directive, into system space.

Syntax

```
void MmLockPagableCodeSection(
   Address
);
```

Parameters

`Address`

Specifies a symbolic address. This address is typically the name of a driver function within a section of driver code that has been marked with something like **#pragma alloc_text (PAGExxxx, driverfunction)**. All functions in the **PAGExxxx** section are then guaranteed to be locked down when this function returns.

Return Value

MmLockPagableCodeSection returns an opaque value that the operating system uses to identify this section of the driver code. This opaque value can be passed subsequently to **MmLockPagableSectionByHandle** (if the driver unlocks and then later relocks the section) or to **MmUnlockPagableImageSection**.

Remarks

The **MmLockPagableCodeSection** routine and **MmUnlockPagableImageSection** (the routine that performs the opposite action) support drivers that can do the following:

- Defer loading a subset of driver routines into resident memory until incoming I/O requests for the driver's devices make it necessary for these routines to process IRPs.
- Make the same subset of driver routines available for paging out when they have completed the processing of I/O requests and no additional requests for the driver's devices are currently expected.

MmLockPagableCodeSection, **MmLockPagableSectionByHandle** and **MmUnlockPagableImageSection** are intended for use by device and intermediate drivers that have the following characteristics:

- The driver has code paths that might not be needed while the system is running, but, if they are needed, the driver's code must be resident because it runs in an arbitrary thread context or at IRQL >= DISPATCH_LEVEL.
- The driver can determine exactly when the pageable routines should be loaded and when they can be paged out again.

For example, the system-supplied fault-tolerant disk driver supports the creation of mirror sets, stripe sets, and volume sets. Yet, a particular machine can be configured only with a mirror set, only with a stripe set, only with a volume set, or with any combination of these three possible options. In these circumstances, the system ftdisk driver reduces the size of its loaded image by marking routines that explicitly support mirror, stripe, and volume sets as belonging to pageable code sections. During driver initialization, pageable code sections are made resident only if the user has configured the

disks to have mirror, stripe, or volume sets. If the user repartitions the disks dynamically, the ftdisk driver locks down any additional pageable code sections necessary to support any mirror, stripe, or volume sets that the user requests.

As other examples, the system-supplied serial and parallel drivers have *DispatchCreate* and *DispatchClose* routines that are called when a particular port is opened for exclusive I/O and when the handle for an opened port is released, respectively. Yet, serial and parallel I/O requests are sporadic, determined by which applications the end user is currently running and which application options the end user is currently exercising. In these circumstances, the system serial and parallel drivers reduce the sizes of their loaded images by marking many routines as belonging to a pageable code section that the *DispatchCreate* routine makes resident only when the first port is opened for I/O.

Note that each of the preceding system drivers satisfies both criteria for having pageable sections: the driver has code paths that might not be needed while the system is running, and the driver can determine exactly when its pageable section should be loaded and can be paged out again.

Because it is an expensive operation to lock down a section, if a driver locks down a pageable code section in more than one place, use **MmLockPagableCodeSection** for the first request. Make subsequent lock requests by calling **MmLockPagableSectionByHandle** passing the handle returned by **MmLockPagableCodeSection**. Locking by handle significantly improves driver performance because the memory manager uses the opaque return value to quickly locate the relevant section rather than searching a loaded module list. A locked down section is unlocked by calling **MmUnlockPagableImageSection**.

Each driver routine within a pageable code section must be marked with the following compiler directive:

```
#pragma alloc_text(PAGExxxx, DriverRoutine)
```

where *xxxx* is an optional four-character, unique identifier for the caller's pageable section and*DriverRoutine* is an entry point to be included within the pageable code section. The keyword **PAGE** and the driver-determined suffix, which can be up to four characters, are case-sensitive; that is, **PAGE** must be capitalized.

A single call to **MmLockPagableCodeSection** in, for example, a driver's *DispatchCreate* routine, causes the entire section, containing every driver routine marked with the same **PAGE***xxxx* identifier, to be locked in system space.

Certain types of driver routines cannot be made part of any driver's pageable section, including the following:

- Never make an ISR pageable. It is possible for a device driver to receive a spurious interrupt even if its device is not in use, particularly if the interrupt vector could be shared. In general, even if a driver can explicitly disable interrupts on its device, an ISR should not be made pageable.
- Never make a DPC routine pageable if the driver cannot control when the DPC is queued, such as any *DpcForIsr* or *CustomDpc* routine that might be queued from an ISR. In general, driver routines that run at IRQL >= DISPATCH_LEVEL and that can be called in an arbitrary thread context or in response to a random external event should not be made pageable.
- Never make the *DispatchRead* or *DispatchWrite* routine pageable in any driver that might be part of the system paging I/O path. The driver of a disk that might contain the system page file must have *DispatchRead* and *DispatchWrite* routines that are resident while the system is running, as must all drivers layered above such a disk driver.

Note that routines in a pageable section marked with the compiler directive **#pragma alloc_text(PAGExxxx, ...)** differ from routines marked with the compiler directive **#pragma alloc_text(INIT, ...)**. The routines in the **INIT** section are not pageable and are discarded as soon as the driver returns from its **DriverEntry** or its *Reinitialize* routine, if it has one.

The memory manager maintains an internal lock count on any driver's pageable section. Calls to **MmLockPagableCodeSection** increment this count and the reciprocal **MmUnlockPagableImageSection** decrements the count. A driver's pageable section is not available to be paged out unless this count is zero.

For more information about creating pageable code sections, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

MmLockPagableDataSection

MmLockPagableSectionByHandle

MmPageEntireDriver

MmResetDriverPaging

MmUnlockPagableImageSection

MmLockPagableDataSection function

1/25/2019 • 2 minutes to read • Edit Online

The **MmLockPagableDataSection** routine locks an entire section of driver data into system space.

Syntax

```
PVOID MmLockPagableDataSection(
  PVOID AddressWithinSection
);
```

Parameters

`AddressWithinSection`

Specifies the symbolic address of one item of data within the pageable section.

Return Value

MmLockPagableDataSection returns an opaque value that identifies the section. This value must be passed subsequently to **MmLockPagableSectionByHandle** or to **MmUnlockPagableImageSection**.

Remarks

Drivers can use this routine, **MmLockPagableSectionByHandle**, and **MmUnlockPagableImageSection** to make their private data that is typically pageable locked into memory.

Data can be locked down if:

- The data is typically accessed at <= APC_LEVEL, but it might need to be accessed at higher IRQL levels for short periods.
- The driver uses the data infrequently and predictably.

For example, drivers for mixer devices use pageable-data sections. Because the driver uses sufficient data to make creating a pageable-data section worthwhile and the driver knows when the data is needed, such a driver uses **MmLockPagableDataSection**, **MmLockPagableSectionByHandle** and **MmUnlockPagableImageSection** to bring a data section into system space when needed and make it available to be paged out when not needed.

A single call to **MmLockPagableDataSection** causes the entire section, containing the referenced data, to be locked into system space.

It is an expensive operation to lock down a section. If a pageable-data section is locked down in more than one place by a driver, use **MmLockPagableDataSection** for the first request. Make subsequent lock requests by calling **MmLockPagableSectionByHandle**, passing the handle returned by **MmLockPagableDataSection**. Locking by handle significantly improves driver performance. A locked down section is unlocked by calling **MmUnlockPagableImageSection**.

The memory manager maintains a reference count on the section. A pageable-data section is only available to be paged out when the reference count is zero. Every lock request increments the count; every unlock request decrements the count. A driver must unlock a section as many times as it locks a section to ensure that such a section will be available to be paged out when the section is not needed. A handle is always valid, no matter what

the count. If the count on a handle is zero and a call is made to **MmLockPagableSectionByHandle**, the count is set to one, and if the section has been paged out, it will be paged in.

Data in a pageable-data section is marked by a compiler directive. To create a pageable data section, use **#pragma data_seg ("PAGE")**, at the beginning of the data module, and **#pragma data_seg ()** at the end of the module. The keyword **PAGE** is case-sensitive, that is, **PAGE** must be capitalized.

Note that there is also a **#pragma data_seg("INIT")** that is used to make data discardable after system initialization. Except for the use of **INIT** rather than **PAGE**, the syntax is the same. However the result is not; use of the **PAGE** directive makes the data section pageable. When the **INIT** directive is used, the data in the section is discarded as soon as the driver returns from its driver entry routine or its reinitialization routine if the driver has one.

For more information about paging data, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmLockPagableCodeSection

MmLockPagableSectionByHandle

MmPageEntireDriver

MmResetDriverPaging

MmUnlockPagableImageSection

MmMapIoSpace function

1/25/2019 • 2 minutes to read • Edit Online

The **MmMapIoSpace** routine maps the given physical address range to nonpaged system space.

Syntax

```
PVOID MmMapIoSpace(
  PHYSICAL_ADDRESS    PhysicalAddress,
  SIZE_T              NumberOfBytes,
  MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`PhysicalAddress`

Specifies the starting physical address of the I/O range to be mapped.

`NumberOfBytes`

Specifies a value greater than zero, indicating the number of bytes to be mapped.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the cache attribute to use to map the physical address range.

Return Value

MmMapIoSpace returns the base virtual address that maps the base physical address for the range. If space for mapping the range is insufficient, it returns **NULL**.

Remarks

A driver must call this routine during device start-up if it receives translated resources of type **CmResourceTypeMemory** in a CM_PARTIAL_RESOURCE_DESCRIPTOR structure. **MmMapIoSpace** maps the physical address returned in the resource list to a logical address through which the driver can access device registers.

For example, drivers of PIO devices that allocate long-term I/O buffers can call this routine to make such buffers accessible or to make device memory accessible.

For more information about using this routine, see Mapping Bus-Relative Addresses to Virtual Addresses.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemory

MmAllocateNonCachedMemory

MmMapLockedPages

MmUnmapIoSpace

MmMapIoSpaceEx function

1/25/2019 • 2 minutes to read • Edit Online

The **MmMapIoSpaceEx** routine maps the given physical address range to non-paged system space using the specified page protection.

Syntax

```
PVOID MmMapIoSpaceEx(
  PHYSICAL_ADDRESS PhysicalAddress,
  SIZE_T           NumberOfBytes,
  ULONG            Protect
);
```

Parameters

`PhysicalAddress`

Specifies the starting physical address of the I/O range to be mapped.

`NumberOfBytes`

Specifies a value greater than zero, indicating the number of bytes to be mapped.

`Protect`

Flag bits that specify the protection to use for the mapped range. The caller must set one of the following flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_READONLY	The mapped range can only be read, not written.
PAGE_READWRITE	The mapped range can be read or written.
PAGE_EXECUTE	The mapped range can be executed, but not read or written.
PAGE_EXECUTE_READ	The mapped range can be executed or read, but not written.
PAGE_EXECUTE_READWRITE	The mapped range can be executed, read, or written.

In addition, the caller can set one (but not both) of the following optional flag bits in the *Protect* parameter.

FLAG BIT	MEANING
PAGE_NOCACHE	Specifies non-cached memory.
PAGE_WRITECOMBINE	Specifies write-combined memory (the memory should not be cached by the processor, but writes to the memory can be combined by the processor).

Return Value

MmMapIoSpaceEx returns the base virtual address that maps the base physical address for the range. If space for mapping the range is insufficient, it returns **NULL**.

Remarks

A driver must call this routine during device start-up if it receives translated resources of type **CmResourceTypeMemory** in a CM_PARTIAL_RESOURCE_DESCRIPTOR structure. **MmMapIoSpaceEx** maps the physical address returned in the resource list to a logical address through which the driver can access device registers.

For example, drivers of PIO devices that allocate long-term I/O buffers can call this routine to make such buffers accessible or to make device memory accessible.

For more information about using this routine, see Mapping Bus-Relative Addresses to Virtual Addresses.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmAllocateContiguousMemory

MmAllocateNonCachedMemory

MmMapLockedPages

MmUnmapIoSpace

MmMapLockedPages function

1/25/2019 • 2 minutes to read • Edit Online

The **MmMapLockedPages** routine is obsolete for Windows 2000 and later versions of Windows, and for Windows Me. It is supported only for WDM drivers that must run on Windows 98. Otherwise, use MmMapLockedPagesSpecifyCache.

The **MmMapLockedPages** routine maps the physical pages that are described by a given MDL.

Syntax

```
PVOID MmMapLockedPages(
  PMDL                                                                    MemoryDescriptorList,
  __drv_strictType(KPROCESSOR_MODE / enum _MODE,__drv_typeConst)KPROCESSOR_MODE AccessMode
);
```

Parameters

`MemoryDescriptorList`

Pointer to an MDL that was updated by MmProbeAndLockPages.

`AccessMode`

Specifies the access mode in which to map the MDL, either **KernelMode** or **UserMode**. Almost all drivers should use **KernelMode**.

Return Value

MmMapLockedPages returns the starting address of the mapped pages. (For NT-based operating systems prior to Windows NT 4.0 Service Pack 4 (SP4), **MmMapLockedPages** returns the beginning address of the first page of the mapped pages.)

Remarks

Use MmUnmapLockedPages to unmap the physical pages that were mapped by **MmMapLockedPages**.

If *AccessMode* is KernelMode and **MmMapLockedPages** cannot map the specified pages, the system issues a bug check. (For this reason, drivers should use MmMapLockedPagesSpecifyCache when available; that routine returns **NULL** on failure, rather than causing a bug check.) If *AccessMode* is **UserMode** and the specified pages cannot be mapped, the routine raises an exception. Callers that specify **UserMode** must wrap the call to **MmMapLockedPages** in a **try/except** block. For more information, see Handling Exceptions.

Callers of **MmMapLockedPages** must be running at IRQL <= DISPATCH_LEVEL if *AccessMode* is **KernelMode**. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Obsolete for Windows 2000 and later versions of Windows, and for Windows Me. This routine is supported only for WDM drivers that must run on Windows 98. Otherwise, use MmMapLockedPagesSpecifyCache.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

MmGetSystemAddressForMdl

MmGetSystemAddressForMdlSafe

MmMapLockedPagesSpecifyCache

MmProbeAndLockPages

MmUnmapLockedPages

MmMapLockedPagesSpecifyCache function

1/25/2019 • 4 minutes to read • Edit Online

The **MmMapLockedPagesSpecifyCache** routine maps the physical pages that are described by an MDL to a virtual address, and enables the caller to specify the cache attribute that is used to create the mapping.

Syntax

```
PVOID MmMapLockedPagesSpecifyCache(
  PMDL                                                                  MemoryDescriptorList,
  __drv_strictType(KPROCESSOR_MODE / enum _MODE,__drv_typeConst)KPROCESSOR_MODE AccessMode,
  __drv_strictTypeMatch(__drv_typeCond)MEMORY_CACHING_TYPE              CacheType,
  PVOID                                                                 RequestedAddress,
  ULONG                                                                 BugCheckOnFailure,
  ULONG                                                                 Priority
);
```

Parameters

`MemoryDescriptorList`

A pointer to the MDL that is to be mapped. This MDL must describe physical pages that are locked down. A locked-down MDL can be built by the MmProbeAndLockPages or MmAllocatePagesForMdlEx routine. For mappings to user space, MDLs that are built by the MmBuildMdlForNonPagedPool routine can be used.

`AccessMode`

Specifies the access mode in which to map the MDL: **KernelMode** or **UserMode**. Almost all drivers should use **KernelMode**.

`CacheType`

Specifies a MEMORY_CACHING_TYPE value, which indicates the cache attribute to use to map the MDL. For more information, see the following Remarks section.

`RequestedAddress`

[in, optional]

If *AccessMode* = **UserMode**, this parameter specifies the starting user virtual address to map the MDL to, or set to **NULL** to allow the system to choose the starting address. The system might round down the requested address to fit address boundary requirements, so callers must check the return value.

`BugCheckOnFailure`

Specifies the behavior of the routine for *AccessMode* = **KernelMode** if the MDL cannot be mapped because of low system resources. If **TRUE**, the system issues a bug check. If **FALSE**, the routine returns **NULL**. Drivers must set this parameter to **FALSE**.

`Priority`

An **MM_PAGE_PRIORITY** value that indicates the importance of success when page table entries (PTEs) are scarce. Starting with Windows 8, the specified priority value can be bitwise-ORed with the **MdlMappingNoWrite** or **MdlMappingNoExecute** flags to specify memory in which writes or instruction execution are disabled. For

more information about the possible values for *Priority*, see MmGetSystemAddressForMdlSafe.

Return Value

MmMapLockedPagesSpecifyCache returns the starting address of the mapped pages. If the pages cannot be mapped and *BugCheckOnFailure* is **FALSE**, the routine returns **NULL**.

Remarks

Use MmUnmapLockedPages to unmap the physical pages that were mapped by **MmMapLockedPagesSpecifyCache**.

If *AccessMode* is **KernelMode**, and if **MmMapLockedPagesSpecifyCache** cannot map the specified pages, the routine returns **NULL** (if *BugCheckOnFailure* = **FALSE**), or the operating system issues a bug check (if *BugCheckOnFailure* = **TRUE**).

If *AccessMode* is **UserMode**, be aware of the following details:

- If the specified pages cannot be mapped, the routine raises an exception. Callers that specify **UserMode** must wrap the call to **MmMapLockedPagesSpecifyCache** in a **try/except** block. For more information, see Handling Exceptions.
- The routine returns a user address that is valid in the context of the process in which the driver is running. For example, if a 64-bit driver is running in the context of a 32-bit application, the buffer is mapped to an address in the 32-bit address range of the application.
- A non-executable mapping is always created when *AccessMode* is **UserMode**. Therefore, using the **MdlMappingNoExecute** flag with the *Priority* parameter is unnecessary in this scenario. However, the **MdlMappingNoWrite** flag can still be used with the *Priority* parameter in this scenario to request a read-only mapping.
- The non-executable protection of the mapping and any write protection of the mapping specified by using the **MdlMappingNoWrite** flag with the *Priority* parameter cannot be changed by code that is running in user mode. For example, if a driver maps some pages into a user process and specifies the **MdlMappingNoWrite** flag, the system guarantees that the process cannot modify the pages.

The routine uses the *CacheType* parameter only if the pages that are described by the MDL do not already have a cache type associated with them. However, in nearly all cases, the pages already have an associated cache type, and this cache type is used by the new mapping. An exception to this rule is for pages that are allocated by MmAllocatePagesForMdl, which do not have a specific cache type associated with them. For such pages, the *CacheType* parameter determines the cache type of the mapping.

A driver must not try to create more than one system-address-space mapping for an MDL. Additionally, because an MDL that is built by the MmBuildMdlForNonPagedPool routine is already mapped to the system address space, a driver must not try to map this MDL into the system address space again by using the **MmMapLockedPagesSpecifyCache** routine (although creating user-address-space mappings is allowed). If it is not known whether a locked-down MDL already has a system-address-space mapping, a driver can use the MmGetSystemAddressForMdlSafe macro instead of **MmMapLockedPagesSpecifyCache**. If the MDL is already mapped into the system address space, **MmGetSystemAddressForMdlSafe** will return the existing system-address-space mapping instead of creating a new mapping.

Warning A driver that maps kernel memory into user address space must avoid exposing potentially sensitive kernel data to untrusted processes. Uninitialized buffers, such as buffers that are allocated from pool, must be explicitly filled with zeros before they are mapped. In addition, the size of a user-mode buffer that is allocated from pool must be a multiple of the virtual memory page size to prevent any part of the pages in the buffer from being used for other allocations. Finally, buffers must not be freed back to the pool while they are still mapped to user address space.

If *AccessMode* is **UserMode**, the caller must be running at IRQL <= APC_LEVEL. If *AccessMode* is **KernelMode**, the caller must be running at IRQL <= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

MmAllocatePagesForMdl

MmAllocatePagesForMdlEx

MmBuildMdlForNonPagedPool

MmGetSystemAddressForMdlSafe

MmProbeAndLockPages

MmUnmapLockedPages

MmMapLockedPagesWithReservedMapping function

1/25/2019 • 2 minutes to read • Edit Online

The **MmMapLockedPagesWithReservedMapping** routine maps all or part of an address range that was previously reserved by the MmAllocateMappingAddress routine.

Syntax

```
PVOID MmMapLockedPagesWithReservedMapping(
  PVOID                                                MappingAddress,
  ULONG                                                PoolTag,
  PMDL                                                 MemoryDescriptorList,
  __drv_strictTypeMatch(__drv_typeCond)MEMORY_CACHING_TYPE CacheType
);
```

Parameters

`MappingAddress`

Pointer to the beginning of the reserved virtual memory range. This must be an address previously returned by MmAllocateMappingAddress.

`PoolTag`

Specifies the pool tag for the reserved memory buffer. This must be identical to the value specified in the *PoolTag* parameter of the call to MmAllocateMappingAddress that reserved the buffer.

`MemoryDescriptorList`

A pointer to the MDL that is to be mapped. This MDL must describe physical pages that are locked down. A locked-down MDL can be built by the MmProbeAndLockPages or MmAllocatePagesForMdlEx routine.

`CacheType`

Specifies the MEMORY_CACHING_TYPE value to use to create the mapping.

Return Value

MmMapLockedPagesWithReservedMapping returns a pointer to the beginning of the mapped memory, or **NULL** if the system could not map the memory. This routine returns **NULL** only if there is an error in the function parameters (for example, the driver's mapping address is not large enough to span the supplied MDL). This function is intended to enable drivers to make forward progress even in low-resource scenarios.

Remarks

The caller can use **MmMapLockedPagesWithReservedMapping** to map a subrange of the virtual memory range reserved by MmAllocateMappingAddress as follows:

1. Use IoAllocateMdl to allocate an MDL. The returned MDL is built using the specified starting address and size of the subrange of the virtual memory range to map.
2. Use MmProbeAndLockPages to lock down the physical pages described by the MDL obtained in step 1.

3. Use **MmMapLockedPagesWithReservedMapping** to actually map the virtual memory to the physical memory that was locked down in step 2. Note that the virtual address returned by this function does include the byte offset that the MDL specifies. However, the **MappedSystemVa** field of the MDL that is set by this function does not include the byte offset.

Once the caller does not need to access the memory, it unmaps the memory with MmUnmapReservedMapping. The caller can map and unmap the memory buffer as needed, and must unmap it prior to freeing the mapping range with MmFreeMappingAddress.

Note that the *MappingAddress* parameter specifies the beginning of the range of memory previously reserved by the caller, not the beginning of the memory subrange to be mapped. The caller specifies the starting address and length of the buffer when it allocates the MDL with IoAllocateMdl. The buffer must fit inside the reserved memory range, but it can be a strict subset.

The routine uses the *CacheType* parameter only if the pages that are described by the MDL do not already have a cache type associated with them. However, in nearly all cases, the pages already have an associated cache type, and this cache type is used by the new mapping. An exception to this rule is for pages that are allocated by MmAllocatePagesForMdl, which do not have a specific cache type associated with them. For such pages, the *CacheType* parameter determines the cache type of the mapping.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoAllocateMdl

MEMORY_CACHING_TYPE

MmAllocateMappingAddress

MmAllocatePagesForMdl

MmAllocatePagesForMdlEx

MmFreeMappingAddress

MmProbeAndLockPages

MmUnmapReservedMapping

MmMapMdl function

1/25/2019 • 2 minutes to read • Edit Online

This function maps physical pages described by a memory descriptor list (MDL) into the system virtual address space.

Syntax

```
NTSTATUS MmMapMdl(
  PMDL            MemoryDescriptorList,
  ULONG           Protection,
  PMM_MDL_ROUTINE DriverRoutine,
  PVOID           DriverContext
);
```

Parameters

`MemoryDescriptorList`

A pointer to a valid MDL.

`Protection`

A bitwise of flags that indicates the protection to set for the pages. Possible values are PAGE_Xxx constants defined in Wdm.h.

`DriverRoutine`

A pointer to a driver-supplied callback routine (MM_MDL_ROUTINE) that is invoked after the MDL is mapped.

`DriverContext`

A pointer to a driver-defined context. The driver's callback function can store any status information in the driver context and then examine the value, when the callback is invoked.

Return Value

If the callback function pointed to by *DriverRoutine* was invoked, this function returns STATUS_SUCCESS. The function returns an appropriate NTSTATUS value if the MDL could not be mapped or the callback function could not be invoked.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows

Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	"<=DISPATCH_LEVEL"

See Also

MM_MDL_ROUTINE

MmPageEntireDriver function

1/25/2019 • 2 minutes to read • Edit Online

The **MmPageEntireDriver** routine causes all of a driver's code and data to be made pageable, overriding the attributes of the various sections that make up the driver's image.

Syntax

```
PVOID MmPageEntireDriver(
  PVOID AddressWithinSection
);
```

Parameters

`AddressWithinSection`

Pointer to a virtual address within the driver (for example, the address of the DriverEntry routine).

Return Value

MmPageEntireDriver returns a pointer to the beginning of the driver image in memory.

Remarks

Use this routine to force a driver to be completely pageable. Drivers that call **MmPageEntireDriver** must not have an InterruptService routine (ISR) registered for any interrupts. If the interrupt occurs while the driver is paged out, the system issues a bug check.

The effect of a call to **MmPageEntireDriver** can be undone by calling MmResetDriverPaging.

If the driver is already completely pageable, calling **MmPageEntireDriver** has no effect. For more information about paging an entire driver, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmLockPagableCodeSection

MmLockPagableDataSection

MmLockPagableSectionByHandle

MmResetDriverPaging

MmUnlockPagableImageSection

MmProbeAndLockPages function

1/25/2019 • 2 minutes to read • Edit Online

The **MmProbeAndLockPages** routine probes the specified virtual memory pages, makes them resident, and locks them in memory.

Syntax

```
void MmProbeAndLockPages(
  PMDL            MemoryDescriptorList,
  KPROCESSOR_MODE AccessMode,
  LOCK_OPERATION  Operation
);
```

Parameters

`MemoryDescriptorList`

A pointer to an MDL that specifies a virtual memory buffer. If the routine successfully locks the pages in memory, the MDL is updated to describe the underlying physical pages.

`AccessMode`

The access mode in which to probe the arguments, either **KernelMode** or **UserMode**.

`Operation`

The type of operation for which the caller wants the access rights probed and the pages locked. Set this parameter to **IoReadAccess**, **IoWriteAccess**, or **IoModifyAccess**. **IoReadAccess** indicates that the driver can examine the contents of the buffer but cannot change the contents. **IoWriteAccess** and **IoModifyAccess**, which are equivalent, indicate that the driver has both read and write access to the buffer.

Return Value

None

Remarks

The highest-level driver in a chain of layered drivers that use direct I/O calls this routine. Drivers that use buffered I/O never call **MmProbeAndLockPages**.

MmProbeAndLockPages performs the following operations:

1. If the specified memory range is paged to a backing store (disk, network, and so on), **MmProbeAndLockPages** makes it resident.
2. The routine then confirms that the pages permit the operation specified by the *Operation* parameter.
3. If the memory range permits the specified operation, the routine locks the pages in memory so that they cannot be paged out. Use the MmUnlockPages routine to unlock the pages.
4. Finally, the routine updates the page frame number (PFN) array in the MDL to describe the locked physical pages.

A successful call to **MmProbeAndLockPages** locks the pages in an MDL and sets the MDL structure to the locked state. Every such call must be matched by a corresponding call to **MmUnlockPages** that unlocks the pages and sets the MDL to the unlocked state. After an **MmProbeAndLockPages** call sets an MDL to the locked state, a second call to MmProbeAndLockPages to lock the same MDL is not allowed until **MmUnlockPages** is first called to unlock the MDL.

If two or more MDLs describe the same physical page, the page can be locked multiple times—once for each MDL. The page is unlocked when the last MDL is set to the unlocked state.

A routine such as MmBuildMdlForNonPagedPool or IoBuildPartialMdl updates an MDL to describe pages that are either nonpageable or already locked. A call to **MmProbeAndLockPages** or **MmUnlockPages** to lock or unlock such an MDL is not allowed.

Calls to **MmProbeAndLockPages** must be enclosed in a **try/except** block. If the pages do not support the specified operation, the routine raises the STATUS_ACCESS_VIOLATION or other exceptions. For more information, see Handling Exceptions.

Callers of **MmProbeAndLockPages** must be running at IRQL <= APC_LEVEL for pageable addresses, or at IRQL <= DISPATCH_LEVEL for nonpageable addresses.

This routine does not provide any guarantees about the virtual address that describes these pages (that is, the virtual address might be unmapped, reused, and so on). However, the physical pages are guaranteed to be locked on successful return.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

IoBuildPartialMdl

MmBuildMdlForNonPagedPool

MmUnlockPages

MmProbeAndLockSelectedPages function

1/25/2019 • 2 minutes to read • Edit Online

The **MmProbeAndLockSelectedPages** routine probes the selected virtual memory pages, makes them resident, and locks them in memory.

Syntax

```
void MmProbeAndLockSelectedPages(
  PMDL                 MemoryDescriptorList,
  PFILE_SEGMENT_ELEMENT SegmentArray,
  KPROCESSOR_MODE      AccessMode,
  LOCK_OPERATION       Operation
);
```

Parameters

`MemoryDescriptorList`

A pointer to a memory descriptor list (MDL) that specifies a virtual memory buffer. If the routine successfully locks the pages in memory, the MDL is updated to describe the underlying physical pages.

`SegmentArray`

A pointer to an array of buffer segments that contain the virtual memory pages.

`AccessMode`

Specifies the access mode in which to probe the arguments, either KernelMode or UserMode.

`Operation`

Specifies the type of operation for which the caller wants the access rights probed and the pages locked. Set this parameter to IoReadAccess, IoWriteAccess, or IoModifyAccess. IoReadAccess indicates that the driver can examine the contents of the buffer but cannot change the contents. IoWriteAccess and IoModifyAccess, which are equivalent, indicate that the driver has both read and write access to the buffer.

Return Value

None

Remarks

Calls to **MmProbeAndLockSelectedPages** must be enclosed in a **try/except** block. If the pages do not support the specified operation, the routine raises the STATUS_ACCESS_VIOLATION or other exceptions. For more information, see Handling Exceptions.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h
IRQL	"<= APC_LEVEL"

See Also

MmProbeAndLockPages

MmProtectMdlSystemAddress function

1/25/2019 • 2 minutes to read • Edit Online

The **MmProtectMdlSystemAddress** routine sets the protection type for a memory address range.

Syntax

```
NTSTATUS MmProtectMdlSystemAddress(
  PMDL  MemoryDescriptorList,
  ULONG NewProtect
);
```

Parameters

`MemoryDescriptorList`

Specifies the memory address range to set the protection type for.

`NewProtect`

Specifies the new protection setting for the memory pages. Drivers should specify one of the following values:

PAGE_NOACCESS

The underlying memory pages cannot be read or written.

PAGE_READONLY

The underlying memory pages can only be read, not written.

PAGE_READWRITE

The underlying memory pages can be read or written.

PAGE_EXECUTE

The underlying memory pages can be executed, but not read or written.

PAGE_EXECUTE_READ

The underlying memory pages can be executed or read, but not written.

PAGE_EXECUTE_READWRITE

The underlying memory pages can be executed, read, or written.

Return Value

MmProtectMdlSystemAddress returns an NTSTATUS code. The possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully changed the protection type for the memory address range.

STA TUS _IN VAL ID_ PA GE_ PR OTE CTI ON	The value specified for *NewProtect* is not a valid one for this routine.
STA TUS _NO T_M APP ED_ VIE W	The MDL has not yet been mapped. **MmProtectMdlSystemAddress** can only be used on MDLs that have already been mapped.

Remarks

The **MmProtectMdlSystemAddress** routine can only be called on an MDL that has already been mapped. For example, the routine can be called on an MDL mapped by MmMapLockedPagesSpecifyCache.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmMapLockedPagesSpecifyCache

MmQuerySystemSize function

1/25/2019 • 2 minutes to read • Edit Online

The **MmQuerySystemSize** routine returns an estimate of the amount of memory in the system.

Syntax

```
MM_SYSTEMSIZE MmQuerySystemSize(

);
```

Parameters

This function has no parameters.

Return Value

MmQuerySystemSize returns one of **MmSmallSystem**, **MmMediumSystem**, or **MmLargeSystem**.

Remarks

This routine can be called during driver initialization to determine how much memory it is practical to allocate for an internal buffer.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

MmIsThisAnNtAsSystem

MmResetDriverPaging function

1/25/2019 • 2 minutes to read • Edit Online

The **MmResetDriverPaging** routine resets the pageable status of a driver's sections to that specified when the driver was compiled.

Syntax

```
void MmResetDriverPaging(
  PVOID AddressWithinSection
);
```

Parameters

`AddressWithinSection`

A pointer to a virtual address in the driver (for example, the address of the DriverEntry routine).

Return Value

None

Remarks

MmResetDriverPaging causes those routines that would not normally be pageable, to be locked into memory. Hence, image sections such as .text and .data will be locked in memory if this routine is called.

A driver that calls this routine must do so before enabling interrupts on its device.

A call to MmPageEntireDriver is not a prerequisite to calling this routine. However, calls to **MmResetDriverPaging** do nothing if the driver's image-section attributes have never been overridden by a call to **MmPageEntireDriver**.

For more information about paging an entire driver, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmLockPagableCodeSection

MmLockPagableDataSection

MmLockPagableSectionByHandle

MmPageEntireDriver

MmUnlockPagableImageSection

MmSizeOfMdl function

1/25/2019 • 2 minutes to read • Edit Online

The **MmSizeOfMdl** routine returns the number of bytes to allocate for an MDL describing a given address range.

Syntax

```
SIZE_T MmSizeOfMdl(
  PVOID  Base,
  SIZE_T Length
);
```

Parameters

`Base`

Pointer to the base virtual address for the range.

`Length`

Supplies the size, in bytes, of the range.

Return Value

MmSizeOfMdl returns the number of bytes required to contain the MDL.

Remarks

Memory for the MDL itself must be allocated from nonpaged pool.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

MmCreateMdl

MmUnlockPagableImageSection function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnlockPagableImageSection** routine releases a section of driver code or driver data, previously locked into system space with **MmLockPagableCodeSection**, **MmLockPagableDataSection** or **MmLockPagableSectionByHandle**, so the section can be paged out again.

Syntax

```
void MmUnlockPagableImageSection(
  PVOID ImageSectionHandle
);
```

Parameters

`ImageSectionHandle`

Specifies the handle returned by a call to MmLockPagableCodeSection or MmLockPagableDataSection.

Return Value

None

Remarks

The handle for a driver's pageable section must not be released if the driver has any outstanding IRPs in its device queue(s) or internal queue(s). A call to **MmUnlockPagableImageSection** restores the pageability of that entire section when there are no more references to the handle for that section.

The memory manager maintains the reference count on the handle to a section. A pageable section is only available to be paged out when the reference count is zero. Every lock request increments the count; every unlock request decrements the count. A driver must unlock a section as many times as it locks a section to make the section available to be paged out.

A handle is always valid, no matter what the count. If the count on a handle is zero and a call is made to **MmLockPagableSectionByHandle**, the count is set to one, and if the section has been paged out, it will be paged in.

In most cases, **MmUnlockPagableImageSection** is called before a driver's *Unload* routine. That is, a driver with a pageable section is likely to have its *DispatchClose* and/or *DispatchShutdown* routine call **MmUnlockPagableImageSection** before its *Unload* routine is called. However, care should be taken in unloadable drivers to release any pageable section before the driver itself is unloaded from the system.

For more information about paging code and data, see Making Drivers Pageable.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"
DDI compliance rules	IrqlMmApcLte, HwStorPortProhibitedDDIs

See Also

MmLockPagableCodeSection

MmLockPagableDataSection

MmLockPagableSectionByHandle

MmPageEntireDriver

MmResetDriverPaging

MmUnlockPages function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnlockPages** routine unlocks the physical pages that are described by the specified memory descriptor list (MDL).

Syntax

```
void MmUnlockPages(
  PMDL MemoryDescriptorList
);
```

Parameters

`MemoryDescriptorList`

A pointer to an MDL.

Return Value

None

Remarks

The memory described by the specified MDL must have been locked previously by a call to MmProbeAndLockPages. If the specified MDL is mapped to system address space, **MmUnlockPages** releases this mapping before it unlocks the pages.

For more information, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

MmProbeAndLockPages

MmUnmapIoSpace function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnmapIoSpace** routine unmaps a specified range of physical addresses previously mapped by **MmMapIoSpace**.

Syntax

```
void MmUnmapIoSpace(
  PVOID  BaseAddress,
  SIZE_T NumberOfBytes
);
```

Parameters

`BaseAddress`

Pointer to the base virtual address to which the physical pages were mapped.

`NumberOfBytes`

Specifies the number of bytes that were mapped.

Return Value

None

Remarks

If a driver calls **MmMapIoSpace** during device start-up, it must call **MmUnmapIoSpace** when it receives a PnP stop-device or remove-device IRP for the same device object.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmMapIoSpace

MmUnmapLockedPages function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnmapLockedPages** routine releases a mapping that was set up by a preceding call to the MmMapLockedPages or MmMapLockedPagesSpecifyCache routine.

Syntax

```
void MmUnmapLockedPages(
  PVOID BaseAddress,
  PMDL  MemoryDescriptorList
);
```

Parameters

`BaseAddress`

Pointer to the base virtual address to which the physical pages were mapped.

`MemoryDescriptorList`

Pointer to an MDL.

Return Value

None

Remarks

Callers of **MmUnmapLockedPages** must be running at IRQL <= DISPATCH_LEVEL if the pages were mapped to system space. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Note that if the call to **MmMapLockedPages** or **MmMapLockedPagesSpecifyCache** specified user mode, the caller must be in the context of the original process before calling **MmUnmapLockedPages**. This is because the unmapping operation occurs in the context of the calling process, and, if the context is incorrect, the unmapping operation could delete the address range of a random process.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	See Remarks section.
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

MmMapLockedPages

MmMapLockedPagesSpecifyCache

MmUnmapReservedMapping function

1/25/2019 • 2 minutes to read • Edit Online

The **MmUnmapReservedMapping** routine unmaps a memory buffer that was mapped by the MmMapLockedPagesWithReservedMapping routine.

Syntax

```
void MmUnmapReservedMapping(
  PVOID BaseAddress,
  ULONG PoolTag,
  PMDL  MemoryDescriptorList
);
```

Parameters

`BaseAddress`

Pointer to the beginning of the reserved virtual memory range. This must be an address returned by MmMapLockedPagesWithReservedMapping.

`PoolTag`

Specifies the pool tag for the reserved memory buffer. This must be identical to the value specified in the *PoolTag* parameter of the call to MmAllocateMappingAddress that initially reserved the buffer.

`MemoryDescriptorList`

Pointer to the MDL that describes the physical memory mapping.

Return Value

None

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

MmAllocateMappingAddress

MmFreeMappingAddress

MmMapLockedPagesWithReservedMapping

NtCommitComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitComplete** routine notifies KTM that the calling resource manager has finished committing a transaction's data.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCommitComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwCommitComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.
STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwCommitComplete** after it has finished servicing a TRANSACTION_NOTIFY_COMMIT notification.

For more information about **ZwCommitComplete**, see Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtCommitEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitEnlistment** routine initiates the commit operation for a specified enlistment's transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCommitEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwCommitEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.
STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_COMMIT_COMPLETE notifications.
STATUS_TRANSACTION_REQUEST_NOT_VALID	The enlistment's transaction is not in a state that allows it to be committed.

STA TUS _TR AN SAC TIO N_N OT_ ACT IVE	The commit operation for this transaction has already been started.
STA TUS _TR AN SAC TIO N_A LRE AD Y_A BO RTE D	The transaction cannot be committed because it has been rolled back.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwCommitEnlistment**.

Callers of **ZwCommitEnlistment** must register to receive TRANSACTION_NOTIFY_COMMIT_COMPLETE notifications.

The **ZwCommitEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_COMMIT notifications to all resource managers that have enlisted in the transaction.

For more information about **ZwCommitEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtCommitTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitTransaction** routine initiates a commit operation for a specified transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCommitTransaction(
  HANDLE  TransactionHandle,
  BOOLEAN Wait
);
```

Parameters

`TransactionHandle`

A handle to a transaction object. Your component receives this handle from ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_COMMIT access to the object.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is **TRUE**, the call returns after the commit operation is complete.

Return Value

ZwCommitTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that was specified for the *TransactionHandle* parameter is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The specified transaction object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the transaction object.
STA TUS _TR AN SAC TIO N_S UPE RIO R_E XIS TS	The caller cannot commit the transaction because a superior transaction manager exists.
STA TUS _TR AN SAC TIO N_A LRE AD Y_A BO RTE D	The transaction cannot be committed because it has been rolled back.
STA TUS _TR AN SAC TIO N_A LRE AD Y_C OM MIT TED	The transaction is already committed.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The commit operation for this transaction has already been started.
STA TUS _PE NDI NG	Commit notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

For more information about how transaction clients should use the **ZwCommitTransaction** routine, see Creating a Transactional Client.

For more information about commit operations, see Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitTransaction

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

NtCreateEnlistment function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateEnlistment** routine creates a new enlistment object for a transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateEnlistment(
  PHANDLE            EnlistmentHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             ResourceManagerHandle,
  HANDLE             TransactionHandle,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              CreateOptions,
  NOTIFICATION_MASK  NotificationMask,
  PVOID              EnlistmentKey
);
```

Parameters

`EnlistmentHandle`

A pointer to a caller-allocated variable that receives a handle to the new enlistment object if the call to **ZwCreateEnlistment** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the enlistment object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following access right flags for enlistment objects:

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
ENLISTMENT_QUERY_INFORMATION	Query information about the enlistment (see ZwQueryInformationEnlistment).
ENLISTMENT_SET_INFORMATION	Set information for the enlistment (see ZwSetInformationEnlistment).
ENLISTMENT_RECOVER	Recover the enlistment (see ZwRecoverEnlistment).
ENLISTMENT_SUBORDINATE_RIGHTS	Perform operations that a resource manager that is not superior performs (see ZwRollbackEnlistment, ZwPrePrepareComplete, ZwPrepareComplete, ZwCommitComplete, ZwRollbackComplete, ZwSinglePhaseReject, ZwReadOnlyEnlistment).
ENLISTMENT_SUPERIOR_RIGHTS	Perform operations that a superior transaction manager must perform (see ZwPrepareEnlistment, ZwPrePrepareEnlistment, ZwCommitEnlistment).

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the ACCESS_MASK

reference page. You can also combine these bitmaps together with additional flags from the previous table. The following table shows how the bitmaps correspond to specific access rights.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
ENLISTMENT_GENERIC_READ	STANDARD_RIGHTS_READ and ENLISTMENT_QUERY_INFORMATION
ENLISTMENT_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, ENLISTMENT_SET_INFORMATION, ENLISTMENT_RECOVER, ENLISTMENT_REFERENCE, ENLISTMENT_SUBORDINATE_RIGHTS, and ENLISTMENT_SUPERIOR_RIGHTS
ENLISTMENT_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, ENLISTMENT_RECOVER, ENLISTMENT_SUBORDINATE_RIGHTS, and ENLISTMENT_SUPERIOR_RIGHTS
ENLISTMENT_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, ENLISTMENT_GENERIC_READ, ENLISTMENT_GENERIC_WRITE, and ENLISTMENT_GENERIC_EXECUTE

`ResourceManagerHandle`

A handle to the caller's resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager.

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. KTM adds this transaction to the list of transactions that the calling resource manager is handling.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`CreateOptions`

Enlistment option flags. The following table contains the only available flag.

CREATEOPTIONS FLAG	MEANING
ENLISTMENT_SUPERIOR	The caller is enlisting as a superior transaction manager for the specified transaction.

This parameter is optional and can be zero.

`NotificationMask`

A bitwise OR of TRANSACTION_NOTIFY_*XXX* values that are defined in Ktmtypes.h. This mask specifies the types of transaction notifications that KTM sends to the caller.

`EnlistmentKey`

A pointer to caller-defined information that uniquely identifies the enlistment. The resource manager receives this pointer when it calls ZwGetNotificationResourceManager or when KTM calls the ResourceManagerNotification

callback routine. The resource manager can maintain a reference count for this key by calling TmReferenceEnlistmentKey and TmDereferenceEnlistmentKey. This parameter is optional and can be **NULL**.

Return Value

ZwCreateEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_INVALID_PARAMETER	The *CreateOptions* or *NotificationMask* parameter's value is invalid, or KTM could not find the transaction that the *TransactionHandle* parameter specifies.
STATUS_INSUFFICIENT_RESOURCES	A memory allocation failed.
STATUS_TRANSACTION_MANAGER_NOT_ONLINE	The enlistment failed because KTM or the resource manager is not in an operational state.

STATUS_TRANSACTION_NOT_ACTIVE	The enlistment failed because the transaction that the *TransactionHandle* parameter specifies is not active.
STATUS_TRANSACTION_SUPERIOR_EXISTS	The caller tried to register as a superior transaction manager but a superior transaction manager already exists.
STATUS_TM_VOLATILE	The caller is trying to register as a superior transaction manager, but the caller's resource manager object is volatile while the associated transaction manager object is not volatile.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

A resource manager calls **ZwCreateEnlistment** to enlist in a transaction.

Resource managers that are not superior must include the ENLISTMENT_SUBORDINATE_RIGHTS flag in their access mask.

Superior transaction managers must include the ENLISTMENT_SUPERIOR_RIGHTS flag in their access masks. Typically, a superior transaction manager includes code that calls ZwRollbackEnlistment, so it must also include the ENLISTMENT_SUBORDINATE_RIGHTS flag.

A resource manager that calls **ZwCreateEnlistment** must eventually call ZwClose to close the object handle.

Your resource manager can use the *EnlistmentKey* parameter to assign a unique value to each enlistment, such as a

pointer to a data structure that contains information about the enlistment. For example, if the resource manager stores the enlistment object's handle in the structure, the resource manager can do the following:

1. Call ZwGetNotificationResourceManager to obtain a notification.
2. Obtain the enlistment key value from the TRANSACTION_NOTIFICATION structure.
3. Use the enlistment key to find the stored enlistment object handle.
4. Call routines that require the enlistment handle as input, such as ZwCommitComplete or ZwRollbackComplete.

For more information about **ZwCreateEnlistment**, see Creating a Resource Manager and Creating a Superior Transaction Manager.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

TRANSACTION_NOTIFICATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCommitComplete

ZwCommitEnlistment

ZwCreateResourceManager

ZwCreateTransaction

ZwGetNotificationResourceManager

ZwOpenEnlistment

ZwOpenResourceManager

ZwOpenTransaction

ZwPrePrepareComplete

ZwPrePrepareEnlistment

ZwPrepareComplete

ZwPrepareEnlistment

ZwQueryInformationEnlistment

ZwReadOnlyEnlistment

ZwRecoverEnlistment

ZwRollbackComplete

ZwRollbackEnlistment

ZwSetInformationEnlistment

ZwSinglePhaseReject

NtCreateResourceManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwCreateResourceManager** routine creates a resource manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateResourceManager(
  PHANDLE            ResourceManagerHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             TmHandle,
  LPGUID             RmGuid,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              CreateOptions,
  PUNICODE_STRING    Description
);
```

Parameters

`ResourceManagerHandle`

A pointer to a caller-allocated variable that receives a handle to the new resource manager object if the call to **ZwCreateResourceManager** is successful.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the resource manager object. In addition to the access rights that are defined for all kinds of objects (see **ACCESS_MASK**), the caller can specify any of the following access right flags for resource manager objects:

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
RESOURCEMANAGER_ENLIST	Enlist in transactions (see ZwCreateEnlistment).
RESOURCEMANAGER_GET_NOTIFICATION	Receive notifications about the transactions that are associated with this resource manager (see ZwGetNotificationResourceManager).
RESOURCEMANAGER_REGISTER_PROTOCOL	Not used.
RESOURCEMANAGER_QUERY_INFORMATION	Query information about the resource manager (see ZwQueryInformationResourceManager).
RESOURCEMANAGER_SET_INFORMATION	Not used.
RESOURCEMANAGER_RECOVER	Recover the resource manager (see ZwRecoverResourceManager).
RESOURCEMANAGER_COMPLETE_PROPAGATION	Not used.

Alternatively, you can specify one or more of the following generic ACCESS_MASK flags. (The STANDARD_RIGHTS_*Xxx* flags are predefined system values that are used to enforce security on system objects.)

You can also combine these generic flags with additional flags from the preceding table. The following table shows how generic access rights correspond to specific access rights.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
RESOURCEMANAGER_GENERIC_READ	STANDARD_RIGHTS_READ, RESOURCEMANAGER_QUERY_INFORMATION, and SYNCHRONIZE
RESOURCEMANAGER_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, RESOURCEMANAGER_SET_INFORMATION, RESOURCEMANAGER_RECOVER, RESOURCEMANAGER_ENLIST, RESOURCEMANAGER_GET_NOTIFICATION, RESOURCEMANAGER_REGISTER_PROTOCOL, RESOURCEMANAGER_COMPLETE_PROPAGATION, and SYNCHRONIZE
RESOURCEMANAGER_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, RESOURCEMANAGER_RECOVER, RESOURCEMANAGER_ENLIST, RESOURCEMANAGER_GET_NOTIFICATION, RESOURCEMANAGER_COMPLETE_PROPAGATION, and SYNCHRONIZE
RESOURCEMANAGER_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, RESOURCEMANAGER_GENERIC_READ, RESOURCEMANAGER_GENERIC_WRITE, and RESOURCEMANAGER_GENERIC_EXECUTE

`TmHandle`

A handle to a transaction manager object that was obtained by a previous all to ZwCreateTransactionManager or ZwOpenTransactionManager.

`RmGuid`

A pointer to a GUID that KTM will use to identify the resource manager. If this pointer is **NULL**, KTM generates a GUID.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

***CREATEOPTIONS* FLAG**	**MEANING**
RESOURCE_MANAGER_COMMUNICATION	For internal use only.
RESOURCE_MANAGER_VOLATILE	The caller will manage volatile resources. It will be non-persistent and will not perform recovery.

This parameter is optional and can be zero.

`Description`

A pointer to a caller-supplied UNICODE_STRING structure that contains a NULL-terminated string. The string provides a description of the resource manager. KTM stores a copy of the string and includes the string in messages that it writes to the log stream. The maximum string length is MAX_RESOURCEMANAGER_DESCRIPTION_LENGTH. This parameter is optional and can be **NULL**.

Return Value

ZwCreateResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that *TmHandle* specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The handle that *TmHandle* specifies is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the specified transaction manager object.
STATUS_TRANSACTION_OBJECT_EXPIRED	The handle that *TmHandle* specifies is closed.

STATUS_INVALID_PARAMETER	The *CreateOptions*parameter's value is invalid or the *Description* parameter's string is too long.
STATUS_TM_VOLATILE	The *CreateOptions*parameter does not specify RESOURCE_MANAGER_VOLATILE but the transaction manager that *TmHandle* specifies is volatile.
STATUS_OBJECT_NAME_COLLISION	The GUID that *ResourceManagerGuid*specifies already exists.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

A resource manager that calls **ZwCreateResourceManager** must eventually call ZwClose to close the object handle.

For more information about **ZwCreateResourceManager**, see Creating a Resource Manager.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

ZwClose

ZwCreateEnlistment

ZwCreateTransactionManager

ZwGetNotificationResourceManager

ZwOpenResourceManager

ZwOpenTransactionManager

ZwQueryInformationResourceManager

ZwRecoverResourceManager

NtCreateTransaction function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateTransaction** routine creates a transaction object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateTransaction(
  PHANDLE            TransactionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  LPGUID             Uow,
  HANDLE             TmHandle,
  ULONG              CreateOptions,
  ULONG              IsolationLevel,
  ULONG              IsolationFlags,
  PLARGE_INTEGER     Timeout,
  PUNICODE_STRING    Description
);
```

Parameters

`TransactionHandle`

A pointer to a caller-allocated variable that receives a handle to the new transaction object, if the call to **ZwCreateTransaction** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following flags for transaction objects.

ACCESS MASK	ALLOWS THE CALLER TO
TRANSACTION_COMMIT	Commit the transaction (see ZwCommitTransaction).
TRANSACTION_ENLIST	Create an enlistment for the transaction (see ZwCreateEnlistment).
TRANSACTION_PROPAGATE	Do not use.
TRANSACTION_QUERY_INFORMATION	Obtain information about the transaction (see ZwQueryInformationTransaction).
TRANSACTION_ROLLBACK	Roll back the transaction (see ZwRollbackTransaction).
TRANSACTION_SET_INFORMATION	Set information for the transaction (see ZwSetInformationTransaction).

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the ACCESS_MASK reference page. You can also combine these bitmaps with additional flags from the preceding table. The following

table shows how the bitmaps correspond to specific access rights.

RIGHTS BITMAP	SET OF SPECIFIC ACCESS RIGHTS
TRANSACTION_GENERIC_READ	STANDARD_RIGHTS_READ, TRANSACTION_QUERY_INFORMATION, and SYNCHRONIZE
TRANSACTION_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, TRANSACTION_SET_INFORMATION, TRANSACTION_COMMIT, TRANSACTION_ENLIST, TRANSACTION_ROLLBACK, TRANSACTION_PROPAGATE, TRANSACTION_SAVEPOINT, and SYNCHRONIZE
TRANSACTION_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, TRANSACTION_COMMIT, TRANSACTION_ROLLBACK, and SYNCHRONIZE
TRANSACTION_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, TRANSACTION_GENERIC_READ, TRANSACTION_GENERIC_WRITE, and TRANSACTION_GENERIC_EXECUTE
TRANSACTION_RESOURCE_MANAGER_RIGHTS	STANDARD_RIGHTS_WRITE, TRANSACTION_GENERIC_READ, TRANSACTION_SET_INFORMATION, TRANSACTION_ENLIST, TRANSACTION_ROLLBACK, TRANSACTION_PROPAGATE, and SYNCHRONIZE

Typically, a resource manager specifies TRANSACTION_RESOURCE_MANAGER_RIGHTS.

The *DesiredAccess* value cannot be zero.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`Uow`

A pointer to a GUID that KTM uses as the new transaction object's unit of work (UOW) identifier. This parameter is optional and can be **NULL**. If this parameter is **NULL**, KTM generates a GUID and assigns it to the transaction object. For more information, see the following Remarks section.

`TmHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. KTM assigns the new transaction object to the specified transaction manager object. If this parameter is **NULL**, KTM assigns the new transaction object to a transaction manager later, when a resource manager creates an enlistment for the transaction.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

OPTION FLAG	MEANING
TRANSACTION_DO_NOT_PROMOTE	Reserved for future use.

`IsolationLevel`

Reserved for future use. Callers must set this parameter to zero.

`IsolationFlags`

Reserved for future use. Callers should set this parameter to zero.

`Timeout`

A pointer to a time-out value. If the transaction has not been committed by the time specified by this parameter, KTM rolls back the transaction. The time-out value is expressed in system time units (100-nanosecond intervals), and can specify either an absolute time or a relative time. If the value pointed to by *Timeout* is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. This pointer is optional and can be **NULL** if you do not want the transaction to have a time-out value. If *Timeout* = **NULL** or * *Timeout* = 0, the transaction never times out. (You can also use ZwSetInformationTransaction to set a time-out value.)

`Description`

A pointer to a caller-supplied UNICODE_STRING structure that contains a NULL-terminated string. The string provides a description of the transaction. KTM stores a copy of the string and includes the string in messages that it writes to the log stream. The maximum string length is MAX_TRANSACTION_DESCRIPTION_LENGTH. This parameter is optional and can be **NULL**.

Return Value

ZwCreateTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *CreateOptions* parameter contains an invalid flag, the *DesiredAccess* parameter is zero, or the *Description* parameter's string is too long.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).

STA TUS _IN VAL ID_ ACL	A security descriptor contains an invalid access control list (ACL).
STA TUS _IN VAL ID_S ID	A security descriptor contains an invalid security identifier (SID).
STA TUS _OB JEC T_N AM E_E XIS TS	The object name that the *ObjectAttributes* parameter specifies already exists.
STA TUS _OB JEC T_N AM E_IN VAL ID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STA TUS _AC CES S_D ENI ED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

The caller can use the *Uow* parameter to specify a UOW identifier for the transaction object. If the caller does not specify a UOW identifier, KTM generates a GUID and assigns it to the transaction object. The caller can later obtain this GUID by calling ZwQueryInformationTransaction.

Typically, you should let KTM generate a GUID for the transaction object, unless your component communicates with another TPS component that has already generated a UOW identifier for the transaction.

To close the transaction handle, the component that called **ZwCreateTransaction** must call ZwClose. If the last transaction handle closes before any component calls ZwCommitTransaction for the transaction, KTM rolls back the transaction.

For more information about how transaction clients should use **ZwCreateTransaction**, see Creating a Transactional Client.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCommitTransaction

ZwCreateEnlistment

ZwCreateTransactionManager

ZwOpenTransaction

ZwOpenTransactionManager

ZwQueryInformationTransaction

ZwRollbackTransaction

ZwSetInformationTransaction

NtCreateTransactionManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwCreateTransactionManager** routine creates a new transaction manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateTransactionManager(
  PHANDLE            TmHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PUNICODE_STRING    LogFileName,
  ULONG              CreateOptions,
  ULONG              CommitStrength
);
```

Parameters

`TmHandle`

A pointer to a caller-allocated variable that receives a handle to the new transaction manager object.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction manager object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following access right flags for transaction manager objects.

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
TRANSACTIONMANAGER_CREATE_RM	Create a resource manager (see ZwCreateResourceManager).
TRANSACTIONMANAGER_QUERY_INFORMATION	Obtain information about the transaction manager (see ZwQueryInformationTransactionManager and ZwEnumerateTransactionObject). Also required for ZwOpenResourceManager, ZwCreateTransaction, and ZwOpenTransaction.)
TRANSACTIONMANAGER_RECOVER	Recover the transaction manager (see ZwRecoverTransactionManager and ZwRollforwardTransactionManager).
TRANSACTIONMANAGER_RENAME	Not used.
TRANSACTIONMANAGER_SET_INFORMATION	Not used.

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the **ACCESS_MASK** reference page. You can also combine these bitmaps with additional flags from the preceding table. The following table shows how the bitmaps correspond to specific access rights.

RIGHTS BITMAP	SET OF SPECIFIC ACCESS RIGHTS

TRANSACTIONMANAGER_GENERIC_READ	STANDARD_RIGHTS_READ and TRANSACTIONMANAGER_QUERY_INFORMATION
TRANSACTIONMANAGER_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, TRANSACTIONMANAGER_SET_INFORMATION, TRANSACTIONMANAGER_RECOVER, TRANSACTIONMANAGER_RENAME, and TRANSACTIONMANAGER_CREATE_RM
TRANSACTIONMANAGER_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE
TRANSACTIONMANAGER_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, TRANSACTIONMANAGER_GENERIC_READ, TRANSACTIONMANAGER_GENERIC_WRITE, and TRANSACTIONMANAGER_GENERIC_EXECUTE

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of a CLFS log file stream to be associated with the transaction manager object. This parameter must be **NULL** if the *CreateOptions* parameter is TRANSACTION_MANAGER_VOLATILE. Otherwise, this parameter must be non-**NULL**. For more information, see the following Remarks section.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

OPTION FLAG	MEANING
TRANSACTION_MANAGER_VOLATILE	The transaction manager object will be volatile. Therefore, it will not use a log file.
TRANSACTION_MANAGER_COMMIT_DEFAULT	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_VOLUME	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_HIVES	For internal use only.
TRANSACTION_MANAGER_COMMIT_LOWEST	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_RECOVERY	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_PROGRESS	For internal use only.

`CommitStrength`

Reserved for future use. This parameter must be zero.

Return Value

ZwCreateTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of an input parameter is invalid.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).
STATUS_LOG_CORRUPTION_DETECTED	KTM encountered an error while creating or opening the log file.
STATUS_INVALID_ACL	A security descriptor contains an invalid access control list (ACL).
STATUS_INVALID_SID	A security descriptor contains an invalid security identifier (SID).

STATUS_OBJECT_NAME_EXISTS	The object name that the *ObjectAttributes* parameter specifies already exists.
STATUS_OBJECT_NAME_COLLISION	The operating system detected a duplicate object name. The error might indicate that the log stream is already being used.
STATUS_OBJECT_NAME_INVALID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

If the log file stream that the *LogFileName* parameter specifies does not exist, KTM calls CLFS to create the stream. If the stream already exists, KTM calls CLFS to open the stream.

Your TPS component must call ZwRecoverTransactionManager after it has called **ZwCreateTransactionManager**

If your TPS component specifies the TRANSACTION_MANAGER_VOLATILE flag in the *CreateOptions* parameter, all resource managers that are associated with the transaction manager object must specify the RESOURCE_MANAGER_VOLATILE flag when they call ZwCreateResourceManager.

A TPS component that calls **ZwCreateTransactionManager** must eventually call ZwClose to close the object handle.

For more information about how use **ZwCreateTransactionManager**, see Creating a Resource Manager.

NtCreateTransactionManager and **ZwCreateTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwOpenTransactionManager

ZwQueryInformationTransactionManager

ZwRecoverTransactionManager

ZwRollforwardTransactionManager

NtEnumerateTransactionObject function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwEnumerateTransactionObject** routine enumerates the KTM objects on a computer.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtEnumerateTransactionObject(
  HANDLE            RootObjectHandle,
  KTMOBJECT_TYPE    QueryType,
  PKTMOBJECT_CURSOR ObjectCursor,
  ULONG             ObjectCursorLength,
  PULONG            ReturnLength
);
```

Parameters

`RootObjectHandle`

A handle to a KTM object. The routine enumerates the child objects of the specified object. This parameter is optional and can be **NULL**. For more information about valid values for this parameter, see the table in the following Remarks section.

`QueryType`

A KTMOBJECT_TYPE-typed value that identifies the object type to enumerate. For more information about valid values for this parameter, see the table in the following Remarks section.

`ObjectCursor`

A pointer to a caller-allocated buffer that begins with a KTMOBJECT_CURSOR structure. **ZwEnumerateTransactionObject** uses the buffer to store the GUIDs of objects that it finds.

`ObjectCursorLength`

The length, in bytes, of the buffer that *ObjectCursor* points to.

`ReturnLength`

A pointer to a caller-allocated location that receives the number of bytes that **ZwEnumerateTransactionObject** returns in the *ObjectCursor* buffer, including the length of the **KTMOBJECT_CURSOR** structure and the length of all returned GUIDs.

Return Value

ZwEnumerateTransactionObject returns STATUS_SUCCESS if the operation succeeds but the routine has not enumerated all the objects. If there are no more objects to enumerate, the routine returns STATUS_NO_MORE_ENTRIES. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The *QueryType* or *ObjectCursorLength* parameter's value is invalid.
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *RootObjectHandle* parameter specifies is not a handle to a valid KTM object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the objects that are being enumerated.

The routine might return other NTSTATUS values.

Remarks

The following table contains the valid values for the *RootObjectHandle* and *QueryType* parameters.

***QUERYTYPE* PARAMETER**	***ROOTOBJECTHANDLE* PARAMETER**	**OBJECTS ENUMERATED**
KTMOBJECT_TRANSACTION_MANAGER	**NULL**	All transaction manager objects
KTMOBJECT_RESOURCE_MANAGER	A handle to a transaction manager object. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.	All resource manager objects that belong to the specified transaction manager object

KTMOBJECT_ENLISTMENT	A handle to a resource manager object. The handle must have RESOURCEMANAGER_QUERY_INFORMATION access to the object.	All enlistment objects that belong to the specified resource manager object
KTMOBJECT_TRANSACTION	A handle to a transaction manager object. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.	All transaction objects that belong to the specified transaction manager object
KTMOBJECT_TRANSACTION	**NULL**	All transaction objects that belong to all transaction manager objects

Most TPS components do not have to call **ZwEnumerateTransactionObject**, but the routine might be useful if you have to write a debugging utility.

Before your component calls **ZwEnumerateTransactionObject**, it must allocate and zero the buffer that *ObjectCursor* points to. The buffer's GUID array can be large enough to receive one or more elements.

To enumerate all of the KTM objects of the specified type, your component must call **ZwEnumerateTransactionObject** repeatedly until it returns STATUS_NO_MORE_ENTRIES.

Every time that the routine is called, it fills the buffer's GUID array with as many object GUIDs that will fit. After each call, your component can use the KTMOBJECT_CURSOR structure's **ObjectIdCount** member to determine the number of object GUIDs that the routine stored in the array.

NtEnumerateTransactionObject and **ZwEnumerateTransactionObject** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Examples

The following code example shows how to enumerate all of the transaction objects on a computer. In this example, the **KTMOBJECT_CURSOR** structure's GUID array contains only one element, so each call to **ZwEnumerateTransactionObject** returns one GUID. The routine creates a Unicode string from the GUID and displays the string.

```
NTSTATUS Status;
UNICODE_STRING GuidString;
KTMOBJECT_CURSOR Cursor;
ULONG ReturnedBytes;

RtlZeroMemory(&Cursor, sizeof(Cursor));

do {
    Status = ZwEnumerateTransactionObject(
                                          NULL,
                                          KTMOBJECT_TRANSACTION,
                                          &Cursor,
                                          sizeof(Cursor),
                                          &ReturnedBytes
                                          );

    if (Status != STATUS_NO_MORE_ENTRIES) {
        RtlStringFromGUID(
                          &Cursor.ObjectIds[0],
                          &GuidString
                          );
        OutputMessage(GuidString.Buffer);
        OutputMessage(L"\r\n");
        RtlFreeUnicodeString(&GuidString);
    }
} while (Status == STATUS_SUCCESS);
if (Status == STATUS_NO_MORE_ENTRIES) {
    Status = STATUS_SUCCESS;
}
```

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KTMOBJECT_CURSOR

KTMOBJECT_TYPE

Using Nt and Zw Versions of the Native System Services Routines

NtGetNotificationResourceManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwGetNotificationResourceManager** routine retrieves the next transaction notification from a specified resource manager's notification queue.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtGetNotificationResourceManager(
  HANDLE                    ResourceManagerHandle,
  PTRANSACTION_NOTIFICATION TransactionNotification,
  ULONG                     NotificationLength,
  PLARGE_INTEGER            Timeout,
  PULONG                    ReturnLength,
  ULONG                     Asynchronous,
  ULONG_PTR                 AsynchronousContext
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_GET_NOTIFICATION access to the object.

`TransactionNotification`

A pointer to a caller-allocated buffer that receives information about the retrieved notification. The buffer must be large enough to contain a TRANSACTION_NOTIFICATION structure plus additional notification-specific arguments.

`NotificationLength`

The length, in bytes, of the buffer that the *TransactionNotification*parameter points to.

`Timeout`

A pointer to a value that specifies a relative or absolute time, in units of 100 nanoseconds. This pointer is optional and can be **NULL**.

If the pointer is **NULL**, **ZwGetNotificationResourceManager** does not return until a transaction notification is available. If a time value is specified, **ZwGetNotificationResourceManager** returns when a notification is available or after the specified time elapses, whichever comes first.

A negative value specifies a time that is relative to the current system time. For example, a relative time value of five seconds causes **ZwGetNotificationResourceManager** to time out five seconds after it is called.

A positive value specifies an absolute time, which is actually relative to 00:00, January 1, 1601. If an absolute time value is specified, the operating system adds the absolute time value to the time value that represents 00:00, January 1, 1601.

If the caller specifies a zero value (instead of a **NULL** pointer), **ZwGetNotificationResourceManager** returns immediately, whether a notification is available or not.

`ReturnLength`

An optional pointer to a variable. If this pointer is not **NULL**, and if the *NotificationLength* parameter's value is too small, **ZwGetNotificationResourceManager** supplies the required length in the variable and returns STATUS_BUFFER_TOO_SMALL.

`Asynchronous`

A ULONG value that must be zero. **ZwGetNotificationResourceManager** does not support asynchronous notifications. Use TmEnableCallbacks to enable asynchronous notifications.

`AsynchronousContext`

A pointer to a ULONG value. This pointer must be **NULL**.

Return Value

ZwGetNotificationResourceManager returns STATUS_SUCCESS if the operation succeeds and a notification is available. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TIMEOUT	The time-out interval that *Timeout* specifies elapsed before a notification became available.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the resource manager object.
STA TUS _BU FFE R_T OO_ SM ALL	The *NotificationLength* parameter's value is too small.

The routine might return other NTSTATUS values.

Remarks

Use the **ZwGetNotificationResourceManager** routine to obtain notifications synchronously. Use the TmEnableCallbacks routine to enable asynchronous notifications.

The received TRANSACTION_NOTIFICATION structure contains the enlistment key that the resource manager specified when it called ZwCreateEnlistment. You can use the enlistment key to identify the enlistment that the notification applies to.

For more information about the **ZwGetNotificationResourceManager** routine, see Creating a Resource Manager.

NtGetNotificationResourceManager and **ZwGetNotificationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"= PASSIVE_LEVEL"
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_NOTIFICATION

TmEnableCallbacks

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwCreateResourceManager

ZwOpenResourceManager

NtOpenEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenEnlistment** routine obtains a handle to an existing enlistment object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenEnlistment(
  PHANDLE            EnlistmentHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             ResourceManagerHandle,
  LPGUID             EnlistmentGuid,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`EnlistmentHandle`

A pointer to a caller-allocated variable that receives a handle to an enlistment object if the call to **ZwOpenEnlistment** succeeds.

`DesiredAccess`

An ACCESS_MASK-typed value that specifies the requested access to the enlistment object. For more information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateEnlistment. This parameter cannot be zero.

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager.

`EnlistmentGuid`

A pointer to a GUID that identifies the enlistment. For more information, see the following Remarks section.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. This parameter is optional and can be **NULL**.

Return Value

ZwOpenEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.
STATUS_INVALID_PARAMETER	The *DesiredAccess* parameter's value is zero, or the *EnlistmentGuid* parameter's value is **NULL**.
STATUS_ENLISTMENT_NOT_FOUND	The enlistment that the *EnlistmentGuid* parameter specifies does not exist for the resource manager that the *RmHandle* parameter specifies.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

Typically, a TPS component calls **ZwOpenEnlistment** after it receives an enlistment GUID from another TPS component that had previously called ZwCreateEnlistment. Most TPS designs do not require calling **ZwOpenEnlistment**.

A resource manager that calls **ZwOpenEnlistment** must eventually call ZwClose to close the object handle.

For more information about **ZwOpenEnlistment**, see Enlistment Objects.

NtOpenEnlistment and **ZwOpenEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateEnlistment

ZwCreateResourceManager

ZwOpenResourceManager

ZwQueryInformationEnlistment

ZwSetInformationEnlistment

NtOpenResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenResourceManager** routine returns a handle to an existing resource manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenResourceManager(
  PHANDLE            ResourceManagerHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             TmHandle,
  LPGUID             ResourceManagerGuid,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`ResourceManagerHandle`

A pointer to a caller-allocated variable that receives the resource manager handle if the call to **ZwOpenResourceManager** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the resource manager object. For more information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateResourceManager. This parameter cannot be zero.

`TmHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager.

`ResourceManagerGuid`

A pointer to the GUID that identifies the resource manager to open.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. This parameter is optional and can be **NULL**.

Return Value

ZwOpenResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The handle that *TmHandle* specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The handle that *TmHandle* specifies is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the specified transaction manager object.
STATUS_INVALID_PARAMETER	The *DesiredAccess* parameter is zero or the *ResourceManagerGuid* parameter is invalid.
STATUS_RESOURCEMANAGER_NOT_FOUND	The specified resource manager could not be found.

STA TUS _TR AN SAC TIO NM AN AGE R_N OT_ ON LIN E	The specified transaction manager is not online.

The routine might return other NTSTATUS values.

Remarks

Typically, a TPS component calls **ZwOpenResourceManager** after it receives an enlistment GUID from another TPS component that had previously called ZwCreateResourceManager. Most TPS designs do not require calling **ZwOpenResourceManager**.

A resource manager that calls **ZwOpenResourceManager** must eventually call ZwClose to close the object handle.

For more information about **ZwOpenResourceManager**, see KTM Objects.

NtOpenResourceManager and **ZwOpenResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateResourceManager

ZwCreateTransactionManager

ZwOpenTransactionManager

NtOpenTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenTransaction** routine obtains a handle to an existing transaction object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenTransaction(
  PHANDLE            TransactionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  LPGUID             Uow,
  HANDLE             TmHandle
);
```

Parameters

`TransactionHandle`

A pointer to a caller-allocated variable that receives a handle to the transaction object if **ZwOpenTransaction** returns STATUS_SUCCESS.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction object. For information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateTransaction.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. The *ObjectAttributes* parameter is optional and can be **NULL**.

`Uow`

A pointer to a GUID that is a transaction object's unit of work (UOW) identifier. This GUID identifies the transaction object to open.

`TmHandle`

A handle to a transaction manager object. If this parameter is not **NULL**, KTM searches only for transaction objects that belong to the specified transaction manager object. If this parameter is **NULL**, KTM searches all transaction objects.

Return Value

ZwCreateTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The *DesiredAccess* or *Uow* parameter was zero.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_TRANSACTION_NOT_FOUND	KTM could not find the transaction object.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

Typically, a resource manager calls **ZwOpenTransaction** after it receives a transaction UOW from a transactional

client that had previously called ZwCreateTransaction.

For more information about **ZwOpenTransaction**, see Creating a Resource Manager.

NtOpenTransaction and **ZwOpenTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwCommitTransaction

ZwCreateTransaction

ZwQueryInformationTransaction

ZwRollbackTransaction

NtOpenTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenTransactionManager** routine obtains a handle to an existing transaction manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenTransactionManager(
  PHANDLE            TmHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PUNICODE_STRING    LogFileName,
  LPGUID             TmIdentity,
  ULONG              OpenOptions
);
```

Parameters

`TmHandle`

A pointer to a caller-allocated variable that receives a handle to the transaction manager object if **ZwOpenTransactionManager** returns STATUS_SUCCESS.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction manager object. For information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateTransactionManager.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of the log file stream that was created when the transaction manager object was created. For more information, see the *LogFileName* parameter of **ZwCreateTransactionManager**. This parameter is optional and can be **NULL**.

`TmIdentity`

A pointer to a GUID that identifies the transaction manager object. This parameter is optional and can be **NULL**.

`OpenOptions`

This parameter is not used and must be zero.

Return Value

ZwOpenTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of an input parameter is invalid.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).
STATUS_OBJECT_NAME_INVALID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STATUS_LOG_CORRUPTION_DETECTED	KTM encountered an error while creating or opening the log file.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

The caller can identify which transaction manager object to open by using one of the following three techniques:

- Use the *LogFileName* parameter to specify the path and file name of a log file stream that was created when the transaction manager object was created.
- Use the *TmIdentity* parameter to specify the GUID that identifies the transaction manager object.
- Use the *ObjectAttributes* parameter to supply an OBJECT_ATTRIBUTES structure that contains the object name that the caller previously specified to the ZwCreateTransactionManager routine.

You must specify only one of the above-listed parameters (an object name, a log file name, or a GUID) and set the other two parameters to **NULL**.

Your TPS component must call ZwRecoverTransactionManager after it has called **ZwOpenTransactionManager**.

A TPS component that calls **ZwOpenTransactionManager** must eventually call ZwClose to close the object handle.

For more information about how to use **ZwOpenTransactionManager**, see Transaction Manager Objects and Creating a Resource Manager.

NtOpenTransactionManager and **ZwOpenTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateTransactionManager

ZwQueryInformationTransactionManager

NtPowerInformation function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPowerInformation** routine sets or retrieves system power information.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPowerInformation(
  POWER_INFORMATION_LEVEL InformationLevel,
  PVOID                   InputBuffer,
  ULONG                   InputBufferLength,
  PVOID                   OutputBuffer,
  ULONG                   OutputBufferLength
);
```

Parameters

`InformationLevel`

Specifies the requested information level, which indicates the specific power information to be set or retrieved. Currently, the only supported *POWER_INFORMATION_LEVEL* value is **PlatformInformation**.

VALUE	MEANING
PlatformInformation	Information represents the currently supported power capabilities of the system. Information may change as drivers are installed. For example, the installation of legacy device drivers that do not support power management might modify the capabilities of the system.

`InputBuffer`

Pointer to a caller-allocated input buffer. This parameter must be **NULL**, otherwise **ERROR_INVALID_PARAMETER** is returned.

`InputBufferLength`

Size, in bytes, of the buffer at *InputBuffer*. The parameter must be set to zero.

`OutputBuffer`

A pointer to an output buffer. The data type of this buffer depends on the information level requested in the *InformationLevel* parameter. For the **PlatformInformation** level, the only currently supported value, the *OutputBuffer* parameter is required and should be of the POWER_PLATFORM_INFORMATION type.

`OutputBufferLength`

Size, in bytes, of the output buffer. Depending on the information level requested, the buffer may be variably sized. *PlatformInformation*, the only currently supported value, requires a buffer that is the size of a POWER_PLATFORM_INFORMATION structure.

Return Value

Returns STATUS_SUCCESS if the call is successful. If the call fails, possible error codes include the following:

RETURN CODE	DESCRIPTION
STATUS_BUFFER_TOO_SMALL	The output buffer is of insufficient size to contain the data being returned.
STATUS_INVALID_PARAMETER	The **PlatformInformation** information level, which is the only currently supported value, requires no input buffer and must contain an output buffer. The caller either supplied an input buffer or no output buffer.
STATUS_ACCESS_DENIED	The caller had insufficient access rights to perform the requested action.

Remarks

NtPowerInformation and **ZwPowerInformation** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Examples

This example illustrates a valid function call.

C++

```
POWER_PLATFORM_INFORMATION PlatformInfo = {0};
NTSTATUS Result = NtPowerInformation(PlatformInformation, NULL, 0, &PlatformInfo, sizeof(PlatformInfo));
```

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntpoapi.h)
Library	Ntoskrnl.lib
DLL	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

POWER_PLATFORM_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

NtPrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrepareComplete** routine notifies KTM that the calling resource manager has finished preparing a transaction's data.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPrepareComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwPrepareComplete** after it has finished servicing a TRANSACTION_NOTIFY_PREPARE notification.

After a resource manager has called **ZwPrepareComplete**, it cannot roll back the transaction.

For more information about **ZwPrepareComplete**, see Handling Commit Operations.

NtPrepareComplete and **ZwPrepareComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrepareComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrePrepareComplete

NtPrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrepareEnlistment** routine initiates the prepare operation for a specified enlistment's transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPrepareEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_PREPARE_COMPLETE notifications.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment's transaction is not in a state that allows it to enter the prepare phase.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwPrepareEnlistment**.

The **ZwPrepareEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_PREPARE notifications to all resource managers that have enlisted in the transaction.

Callers of **ZwPrepareEnlistment** must register to receive TRANSACTION_NOTIFY_PREPARE_COMPLETE notifications.

For more information about **ZwPrepareEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

NtPrePrepareEnlistment and **ZwPrePrepareEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrepareEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtPrePrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrePrepareComplete** routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPrePrepareComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrePrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwPrePrepareComplete** after it has finished servicing a TRANSACTION_NOTIFY_PREPREPARE notification.

For more information about **ZwPrePrepareComplete**, see Handling Commit Operations.

NtPrePrepareComplete and **ZwPrePrepareComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrePrepareComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

NtPrePrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrePrepareEnlistment** routine initiates the pre-prepare operation for a specified enlistment's transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtPrePrepareEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrePrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_PREPREPARE_COMPLETE notifications.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment's transaction is not in a state that allows it to enter the prepare phase.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwPrePrepareEnlistment**.

Callers of **ZwPrePrepareEnlistment** must register to receive TRANSACTION_NOTIFY_PREPREPARE_COMPLETE notifications.

The **ZwPrePrepareEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_PREPREPARE notifications to all resource managers that have enlisted in the transaction.

For more information about **ZwPrePrepareEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

NtPrepareEnlistment and **ZwPrepareEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrePrepareEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtQueryInformationEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationEnlistment** routine retrieves information about a specified enlistment object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationEnlistment(
  HANDLE                      EnlistmentHandle,
  ENLISTMENT_INFORMATION_CLASS EnlistmentInformationClass,
  PVOID                       EnlistmentInformation,
  ULONG                       EnlistmentInformationLength,
  PULONG                      ReturnLength
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_QUERY_INFORMATION access to the object.

`EnlistmentInformationClass`

An ENLISTMENT_INFORMATION_CLASS-typed enumeration value that specifies the information to be obtained. This value must be one of the following values:

- **EnlistmentBasicInformation**
- **EnlistmentRecoveryInformation**

The enumeration's **EnlistmentFullInformation** value is not used with **ZwQueryInformationEnlistment**.

`EnlistmentInformation`

A pointer to a caller-allocated buffer that receives the information that the *EnlistmentInformationClass*parameter specifies. If the *EnlistmentInformationClass* parameter's value is **EnlistmentBasicInformation**, this buffer's structure type must be ENLISTMENT_BASIC_INFORMATION. If the *EnlistmentInformationClass* parameter's value is **EnlistmentRecoveryInformation**, this buffer's type must match the caller-defined type that the caller used when it called ZwSetInformationEnlistment.

`EnlistmentInformationLength`

The length, in bytes, of the buffer that the *EnlistmentInformation* parameter points to.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *EnlistmentInformation*buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *EnlistmentInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The *EnlistmentInformationLength* parameter's value is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call ZwSetInformationEnlistment to set enlistment-specific recovery information for an enlistment object and then call **ZwQueryInformationEnlistment** to retrieve the recovery information.

For more information about **ZwQueryInformationEnlistment**, see Handling Recovery Operations.

Callers of **ZwQueryInformationEnlistment** must be running at IRQL = PASSIVE_LEVEL.

NtQueryInformationEnlistment and **ZwQueryInformationEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ENLISTMENT_BASIC_INFORMATION

ENLISTMENT_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwSetInformationEnlistment

NtQueryInformationResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationResourceManager** routine retrieves information about a specified resource manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationResourceManager(
  HANDLE                          ResourceManagerHandle,
  RESOURCEMANAGER_INFORMATION_CLASS ResourceManagerInformationClass,
  PVOID                           ResourceManagerInformation,
  ULONG                           ResourceManagerInformationLength,
  PULONG                          ReturnLength
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_QUERY_INFORMATION access to the object.

`ResourceManagerInformationClass`

A RESOURCEMANAGER_INFORMATION_CLASS-typed value that specifies the information to retrieve. This value must be **ResourceManagerBasicInformation**.

`ResourceManagerInformation`

A pointer to a caller-allocated RESOURCEMANAGER_BASIC_INFORMATION structure that receives information from **ZwQueryInformationResourceManager**.

`ResourceManagerInformationLength`

The length, in bytes, of the buffer that the *ResourceManagerInformation* parameter points to.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *ResourceManagerInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *ResourceManagerInformationClass* parameter's value is invalid.
STATUS_BUFFER_TOO_SMALL	The buffer size that the *ResourceManagerInformationLength* parameter specifies is smaller than the RESOURCEMANAGER_BASIC_INFORMATION structure.
STATUS_BUFFER_OVERFLOW	The buffer size that the *ResourceManagerInformationLength* parameter specifies is too small to receive all the variable-length information that is available.

STA **TUS** **_AC** **CES** **S_D** **ENI** **ED**	The caller does not have appropriate access to the resource manager object.

The routine might return other NTSTATUS values.

Remarks

For more information about the **ZwQueryInformationResourceManager** routine, see Creating a Resource Manager.

NtQueryInformationResourceManager and **ZwQueryInformationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

RESOURCEMANAGER_BASIC_INFORMATION

RESOURCEMANAGER_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateResourceManager

ZwOpenResourceManager

ZwRecoverResourceManager

ZwSetInformationResourceManager

NtQueryInformationTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationTransaction** routine retrieves information about a specified transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationTransaction(
  HANDLE                      TransactionHandle,
  TRANSACTION_INFORMATION_CLASS TransactionInformationClass,
  PVOID                       TransactionInformation,
  ULONG                       TransactionInformationLength,
  PULONG                      ReturnLength
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_QUERY_INFORMATION access to the object.

`TransactionInformationClass`

A TRANSACTION_INFORMATION_CLASS-typed value that specifies the information to obtain. The value must be one of the following values:

- **TransactionBasicInformation**
- **TransactionPropertiesInformation**
- **TransactionEnlistmentInformation**

The **TransactionFullInformation** value is not used with **ZwQueryInformationTransaction**.

`TransactionInformation`

A pointer to a caller-allocated buffer that receives the information that the *TransactionInformationClass* parameter specifies. The buffer's structure type must be TRANSACTION_BASIC_INFORMATION, TRANSACTION_PROPERTIES_INFORMATION, or TRANSACTION_ENLISTMENTS_INFORMATION.

`TransactionInformationLength`

The length, in bytes, of the buffer that the *TransactionInformation* parameter points to, including the length of any additional array elements that the caller has allocated to receive information.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *TransactionInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The length of the buffer that is specified by the *TransactionInformationLength* parameter is incorrect.
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle*parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.

STA **TUS** **_BU** **FFE** **R_O** **VER** **FLO** **W**	The buffer that the *TransactionInformation*parameter specifies is too small.

The routine might return other NTSTATUS values.

Remarks

For more information about how to use **ZwQueryInformationTransaction**, see Creating a Transactional Client.

NtQueryInformationTransaction and **ZwQueryInformationTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_BASIC_INFORMATION

TRANSACTION_ENLISTMENTS_INFORMATION

TRANSACTION_INFORMATION_CLASS

TRANSACTION_PROPERTIES_INFORMATION

TmGetTransactionId

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

ZwSetInformationTransaction

NtQueryInformationTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationTransactionManager** routine retrieves information about a specified transaction manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryInformationTransactionManager(
  HANDLE                              TransactionManagerHandle,
  TRANSACTIONMANAGER_INFORMATION_CLASS TransactionManagerInformationClass,
  PVOID                               TransactionManagerInformation,
  ULONG                               TransactionManagerInformationLength,
  PULONG                              ReturnLength
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.

`TransactionManagerInformationClass`

A TRANSACTIONMANAGER_INFORMATION_CLASS-typed enumeration value that specifies the information to be obtained. This value must be one of the following:

- **TransactionManagerBasicInformation**
- **TransactionManagerLogInformation**
- **TransactionManagerLogPathInformation**

The enumeration's **TransactionManagerOnlineProbeInformation** value is not used with **ZwQueryInformationTransactionManager**.

`TransactionManagerInformation`

A pointer to a caller-allocated buffer that receives the information that the *TransactionManagerInformationClass* parameter specifies. The buffer's structure type must be TRANSACTIONMANAGER_BASIC_INFORMATION, TRANSACTIONMANAGER_LOG_INFORMATION, TRANSACTIONMANAGER_LOGPATH_INFORMATION, or TRANSACTIONMANAGER_RECOVERY_INFORMATION.

`TransactionManagerInformationLength`

The length, in bytes, of the buffer that the *TransactionManagerInformation* parameter points to, including the length of any additional array elements that the caller has allocated to receive information.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *TransactionManagerInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionManagerInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The length of the buffer that is specified by the *TransactionManagerInformationLength*parameter is incorrect.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction manager object.

STA **TUS** **_BU** **FFE** **R_T** **OO_** **SM** **ALL**	The buffer that is specified by the *TransactionManagerInformation*parameter is too small.

The routine might return other NTSTATUS values.

Remarks

For more information about how to use **ZwQueryInformationTransactionManager**, see Using Log Streams with KTM.

NtQueryInformationTransactionManager and **ZwQueryInformationTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTIONMANAGER_BASIC_INFORMATION

TRANSACTIONMANAGER_INFORMATION_CLASS

TRANSACTIONMANAGER_LOGPATH_INFORMATION

TRANSACTIONMANAGER_LOG_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRecoverTransactionManager

ZwRollforwardTransactionManager

NtReadOnlyEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwReadOnlyEnlistment** routine sets a specified enlistment to be read-only.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtReadOnlyEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwReadOnlyEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state. For example, an enlistment cannot be read-only if the ENLISTMENT_SUPERIOR flag was specified to ZwCreateEnlistment.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

Your component cannot call **ZwReadOnlyEnlistment** for a superior enlistment.

For more information about **ZwReadOnlyEnlistment**, see Creating a Read-Only Enlistment.

NtReadOnlyEnlistment and **ZwReadOnlyEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtRecoverEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverEnlistment** routine initiates a recovery operation for the transaction that is associated with a specified enlistment.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRecoverEnlistment(
  HANDLE EnlistmentHandle,
  PVOID  EnlistmentKey
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_RECOVER access to the object.

`EnlistmentKey`

A pointer to the enlistment key value that the resource manager previously specified as the *EnlistmentKey* parameter to ZwCreateEnlistment. This parameter is optional and can be **NULL** if the resource manager did not provide an enlistment key when it called **ZwCreateEnlistment**.

Return Value

ZwRecoverEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.

STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_TRANSACTION_NOT_EQUEST_NOT_VALID	The transaction or its enlistment is not in the correct state.
STATUS_PENDING	KTM has added a commit, rollback, or in-doubt notification to the resource manager's notification queue, and the resource manager is not using a ResourceManagerNotification callback routine.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

After a resource manager calls **ZwRecoverEnlistment**, KTM delivers either a TRANSACTION_NOTIFY_COMMIT, TRANSACTION_NOTIFY_ROLLBACK, or TRANSACTION_NOTIFY_INDOUBT notification for the enlistment. Therefore, unless the resource manager is using a ResourceManagerNotification callback routine, it must immediately call ZwGetNotificationResourceManager if **ZwRecoverEnlistment** returns STATUS_SUCCESS or STATUS_PENDING.

For more information about **ZwRecoverEnlistment**, see Handling Recovery Operations.

NtRecoverEnlistment and **ZwRecoverEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine

can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ResourceManagerNotification

TmRecoverEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwGetNotificationResourceManager

ZwOpenEnlistment

NtRecoverResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverResourceManager** routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRecoverResourceManager(
  HANDLE ResourceManagerHandle
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_RECOVER access to the object.

Return Value

ZwRecoverResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STATUS_TRANSACTION_NOT_ENLISTED	The transaction manager that is associated with the specified resource manager is not available.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the resource manager object.

The routine might return other NTSTATUS values.

Remarks

When a resource manager calls **ZwRecoverResourceManager**, KTM sends a TRANSACTION_NOTIFY_RECOVER notification to the resource manager for each of the resource manager's enlistments. (If the enlistment is a superior enlistment, KTM sends TRANSACTION_NOTIFY_RECOVER_QUERY instead of TRANSACTION_NOTIFY_RECOVER.)

The final notification that KTM sends to the resource manager is TRANSACTION_NOTIFY_LAST_RECOVER.

For more information about **ZwRecoverResourceManager**, see Handling Recovery Operations.

NtRecoverResourceManager and **ZwRecoverResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

TmRecoverResourceManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateResourceManager

ZwOpenResourceManager

ZwQueryInformationResourceManager

ZwSetInformationResourceManager

NtRecoverTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverTransactionManager** routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRecoverTransactionManager(
  HANDLE TransactionManagerHandle
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_RECOVER access to the object.

Return Value

ZwRecoverTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the transaction manager object.
STA TUS _TM _VO LAT ILE	The transaction manager object is volatile and, therefore, its transactions cannot be recovered.
STA TUS _UN SUC CES SFU L	The transaction manager object is not in a state that allows recovery.

The routine might return other NTSTATUS values.

Remarks

The **ZwRecoverTransactionManager** routine tries to reconstruct the state of the transaction manager object, including all transactions, enlistments, and resource managers, from the recovery information that is in the object's log stream, beginning at the last restart area that KTM created and ending at the stream's end.

To recover up to a specific virtual time, use the ZwRollforwardTransactionManager routine instead of **ZwRecoverTransactionManager**.

Your TPS component must call **ZwRecoverTransactionManager** after it has called ZwCreateTransactionManager or ZwOpenTransactionManager.

For more information about recovery operations, see Handling Recovery Operations.

NtRecoverTransactionManager and **ZwRecoverTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRecoverTransactionManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRollforwardTransactionManager

NtRenameTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **NtRenameTransactionManager** routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRenameTransactionManager(
  PUNICODE_STRING LogFileName,
  LPGUID          ExistingTransactionManagerGuid
);
```

Parameters

`LogFileName`

`ExistingTransactionManagerGuid`

Return Value

None

Requirements

Target Platform	Windows
Header	wdm.h
IRQL	"<=APC_LEVEL"

See Also

TmRenameTransactionManager

NtRollbackComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackComplete** routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRollbackComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwRollbackComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwRollbackComplete** after it has finished servicing a TRANSACTION_NOTIFY_ROLLBACK notification.

For more information about **ZwRollbackComplete**, see Handling Rollback Operations.

NtRollbackComplete and **ZwRollbackComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRollbackComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

NtRollbackEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackEnlistment** routine rolls back the transaction that is associated with a specified enlistment.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRollbackEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwRollbackEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment cannot be rolled back.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call **ZwRollbackEnlistment** to roll back a transaction at any time before it calls ZwPrepareComplete.

For more information about **ZwRollbackEnlistment**, see Handling Rollback Operations.

NtRollbackEnlistment and **ZwRollbackEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRollbackEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

NtRollbackTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackTransaction** routine initiates a rollback operation for a specified transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRollbackTransaction(
  HANDLE  TransactionHandle,
  BOOLEAN Wait
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_ROLLBACK access to the object.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is set to **TRUE**, the call does not return until the rollback operation is complete.

Return Value

ZwRollbackTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle* parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.
STATUS_TRANSACTION_ALREADY_COMMITTED	The transaction cannot be rolled back because it has already been committed.
STATUS_TRANSACTION_NOT_ROLLBACK	The transaction has not been committed but its current state does not permit rollback.
STATUS_PENDING	Rollback notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

For more information about **ZwRollbackTransaction**, see Handling Rollback Operations.

NtRollbackEnlistment and **ZwRollbackEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCommitTransaction

ZwCreateTransaction

ZwOpenTransaction

ZwQueryInformationTransaction

NtRollforwardTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollforwardTransactionManager** routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtRollforwardTransactionManager(
  HANDLE         TransactionManagerHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_RECOVER access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**. For more information about this parameter, see the following Remarks section.

Return Value

ZwRollforwardTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.

STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction manager object.
STATUS_TM_VOLATILE	The specified transaction manager object does not have a log file, so recovery is not available.
STATUS_UNSUCCESSFUL	The specified transaction manager object is in an unexpected state.

The routine might return other NTSTATUS values.

Remarks

The **ZwRollforwardTransactionManager** routine recovers all logged activity that KTM finds in the transaction manager's log file, up to and including the virtual clock value that the *TmVirtualClock* parameter supplies.

Your component can traverse the log file incrementally by calling **ZwRollforwardTransactionManager** repetitively and setting the *VirtualClock* parameter to a higher value before each call.

If the *TmVirtualClock* parameter is **NULL**, calling **ZwRollforwardTransactionManager** is equivalent to calling ZwRecoverTransactionManager.

For more information about recovery operations, see Handling Recovery Operations.

NtRollforwardTransactionManager and **ZwRollforwardTransactionManager** are two versions of the same Windows Native System Services routine. The **NtRollforwardTransactionManager** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwRollforwardTransactionManager** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRecoverTransactionManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRecoverTransactionManager

NtSetInformationEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationEnlistment** routine sets information for a specified enlistment object.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationEnlistment(
  HANDLE                      EnlistmentHandle,
  ENLISTMENT_INFORMATION_CLASS EnlistmentInformationClass,
  PVOID                       EnlistmentInformation,
  ULONG                       EnlistmentInformationLength
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SET_INFORMATION access to the object.

`EnlistmentInformationClass`

A ENLISTMENT_INFORMATION_CLASS-typed enumeration value that specifies the type of information to be set. This value must be **EnlistmentRecoveryInformation**.

The enumeration's **EnlistmentBasicInformation** and **EnlistmentFullInformation** values are not used with **ZwSetInformationEnlistment**.

`EnlistmentInformation`

A pointer to a caller-allocated buffer that contains caller-defined recovery information for the enlistment.

`EnlistmentInformationLength`

The length, in bytes, of the buffer that the *EnlistmentInformation* parameter points to.

Return Value

ZwSetInformationEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *EnlistmentInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The *EnlistmentInformationLength* parameter's value is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can use the **ZwSetInformationEnlistment** routine to set recovery information for an enlistment. KTM writes the recovery information to the log stream, and the resource manager can call

ZwQueryInformationEnlistment to read this information from the log stream at any time.

Each subsequent call to **ZwSetInformationEnlistment** deletes the recovery information that the previous call specified before it stores the new recovery information.

For more information about how to use **ZwSetInformationEnlistment**, see Using Log Streams with KTM.

NtSetInformationEnlistment and **ZwSetInformationEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ENLISTMENT_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwQueryInformationEnlistment

NtSetInformationResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationResourceManager** routine is not used.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationResourceManager(
  HANDLE                          ResourceManagerHandle,
  RESOURCEMANAGER_INFORMATION_CLASS ResourceManagerInformationClass,
  PVOID                           ResourceManagerInformation,
  ULONG                           ResourceManagerInformationLength
);
```

Parameters

`ResourceManagerHandle`

Not used.

`ResourceManagerInformationClass`

Not used.

`ResourceManagerInformation`

Not used.

`ResourceManagerInformationLength`

Not used.

Return Value

This function does not return a value.

Remarks

NtSetInformationResourceManager and **ZwSetInformationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Target Platform	Windows

Header	wdm.h

See Also

Using Nt and Zw Versions of the Native System Services Routines

NtSetInformationTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationTransaction** routine sets information for a specified transaction.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationTransaction(
  HANDLE                      TransactionHandle,
  TRANSACTION_INFORMATION_CLASS TransactionInformationClass,
  PVOID                       TransactionInformation,
  ULONG                       TransactionInformationLength
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_SET_INFORMATION access to the object.

`TransactionInformationClass`

A TRANSACTION_INFORMATION_CLASS-typed value that specifies the type of information to set. The value must be **TransactionPropertiesInformation**.

`TransactionInformation`

A pointer to a caller-allocated buffer that contains the information to set. The buffer's structure type must be TRANSACTION_PROPERTIES_INFORMATION.

`TransactionInformationLength`

The length, in bytes, of the buffer that the *TransactionInformation* parameter points to.

Return Value

ZwSetInformationTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionInformationClass* parameter's value is invalid.

STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle* parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.
STATUS_INFO_LENGTH_MISMATCH	The *TransactionInformationLength*parameter's value is invalid.
STATUS_INVALID_PARAMETER	The contents of the buffer that the *TransactionInformation* buffer specifies is invalid.

The routine might return other NTSTATUS values.

Remarks

For more information about **ZwSetInformationTransaction**, see Creating a Transactional Client.

NtSetInformationTransaction and **ZwSetInformationTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_INFORMATION_CLASS

TRANSACTION_PROPERTIES_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

ZwQueryInformationTransaction

NtSetInformationTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

Do not call this routine from kernel-mode code.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationTransactionManager(
  HANDLE                              TmHandle,
  TRANSACTIONMANAGER_INFORMATION_CLASS TransactionManagerInformationClass,
  PVOID                               TransactionManagerInformation,
  ULONG                               TransactionManagerInformationLength
);
```

Parameters

`TmHandle`

`TransactionManagerInformationClass`

`TransactionManagerInformation`

`TransactionManagerInformationLength`

Return Value

See the **Zw*Xxx*** equivalent.

Do not call this routine from kernel-mode code; instead, call the **Zw*Xxx*** equivalent. User-mode code can call this routine or the **Zw*Xxx*** equivalent. For further comments, if any, see the ZwXxx equivalent

The driver must check the return value of the function.

Requirements

Target Platform	Windows
Header	wdm.h
IRQL	"<=APC_LEVEL"

See Also

See the **Zw*Xxx*** equivalent.

NtSinglePhaseReject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSinglePhaseReject** routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.

Syntax

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSinglePhaseReject(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwSinglePhaseReject returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call **ZwSinglePhaseReject** when it receives a TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notification.

For more information about **ZwSinglePhaseReject**, see Handling Commit Operations.

NtSinglePhaseReject and **ZwSinglePhaseReject** are two versions of the same Windows Native System Services routine. The **NtSinglePhaseReject** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwSinglePhaseReject** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmSinglePhaseReject

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ObCloseHandle function

1/25/2019 • 3 minutes to read • Edit Online

The **ObCloseHandle** routine closes an object handle.

Syntax

```
NTSTATUS ObCloseHandle(
  HANDLE          Handle,
  KPROCESSOR_MODE PreviousMode
);
```

Parameters

`Handle`

A handle to a system-supplied object of any type.

`PreviousMode`

Specifies the previous processor mode of the thread that opened the handle. To close a kernel handle, set this parameter to **KernelMode**. To close a *user handle*, set this parameter to **UserMode**. For more information about these two handle types, see Remarks.

Return Value

ObCloseHandle returns STATUS_SUCCESS if the call is successful. Possible error return values include the following NTSTATUS codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_HANDLE	*Handle* is not a valid handle.
STATUS_HANDLE_NOT_CLOSABLE	The calling thread does not have permission to close the handle.

Remarks

A kernel-mode driver calls **ObCloseHandle** to close a handle to any type of object that is created by the Windows kernel. A driver must close every handle that it opens as soon as the handle is no longer required.

After **ObCloseHandle** closes an object's handle, the caller must treat the handle as invalid and avoid using the handle to access the object. However, other handles might remain open on the same object. During an **ObCloseHandle** call, the system decrements the handle count for the object and checks whether the object can be deleted. The system does not delete the object until all of the object's handles are closed and all reference-counted pointers to the object are released.

The *PreviousMode* parameter specifies whether the handle to be closed is a kernel handle or a user handle. To close a kernel handle, set *PreviousMode* to **KernelMode**. To close a user handle, set *PreviousMode* to **UserMode**.

A kernel handle is a handle that is opened by a system thread, or by a kernel-mode driver that assigns the OBJ_KERNEL_HANDLE attribute to the handle. (For example, see the description of OBJ_KERNEL_HANDLE in ZwCreateFile.) If a kernel-mode driver opens a handle for its private use, and this driver runs in the context of a user-mode thread, the driver must open the handle with the OBJ_KERNEL_HANDLE attribute. This attribute ensures that the handle is inaccessible to user-mode applications.

A user handle is a handle that is opened by a user-mode application, or by a kernel-mode driver that runs in the context of a user-mode thread but that does not open the handle with the OBJ_KERNEL_HANDLE attribute. If a driver creates a user handle to be used by a user-mode application, but an error occurs that requires the driver to close the handle on behalf of the application, the driver can call **ObCloseHandle** to close the handle.

The **ZwClose** routine is similar to **ObCloseHandle** but can close only kernel handles. The call **ZwClose**(*hObject*), which closes kernel handle *hObject*, has the same effect as the call **ObCloseHandle**(*hObject*, **KernelMode**). For more information about closing a kernel handle, see ZwClose.

To determine whether a handle is a kernel handle or a user handle, a driver that receives a handle can call the ExGetPreviousMode routine. Or, the driver can read the **RequestorMode** field from the IRP structure that describes the I/O request. The I/O manager sets the **RequestorMode** field to the previous processor mode of the thread that requested the I/O operation.

Callers of **ObCloseHandle** should not assume that this routine automatically waits for all pending I/O operations to complete before it returns.

For more information, see Object Handles.

ObCloseHandle is not declared in a header file prior to Windows 7. To use this routine in your driver, include the following function declaration in your driver code:

```
#if (NTDDI_VERSION < NTDDI_WIN7)
NTKERNELAPI
NTSTATUS
  ObCloseHandle(
    __in HANDLE Handle,
    __in KPROCESSOR_MODE PreviousMode
    );
#endif
```

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ExGetPreviousMode

IRP

ZwClose

ZwCreateFile

ObDereferenceObject macro

1/25/2019 • 2 minutes to read • Edit Online

The **ObDereferenceObject** routine decrements the given object's reference count and performs retention checks.

Syntax

```
void ObDereferenceObject(
   a
);
```

Parameters

`a`

Pointer to the object's body.

Return Value

ObDereferenceObject returns a value that is reserved for system use. Drivers must treat this value as VOID.

Remarks

ObDereferenceObject decreases the reference count of an object by one. If the object was created as temporary (the OBJ_PERMANENT flag was not specified on creation), and the reference count reaches zero, the object can be deleted by the system.

A driver can delete a temporary object it created by decreasing its reference count to zero. A driver must never attempt to delete an object it did not create.

An object is permanent if it was created with the OBJ_PERMANENT object attribute flag specified. (For more information about object attributes, see InitializeObjectAttributes.) A permanent object is created with a reference count of one, so it is not deleted when the driver dereferences it. A driver can only delete a permanent object it created by using the ZwMakeTemporaryObject routine to make it temporary. Use the following steps to delete a permanent object that you created:

1. Call **ObDereferenceObject**.
2. Call the appropriate **ZwOpen*Xxx*** or **ZwCreate*Xxx*** routine to get a handle for the object, if necessary.
3. Call ZwMakeTemporaryObject with the handle obtained in step 2.
4. Call ZwClose with the handle obtained in step 2.

> **Important**
> Use ObDereferenceObjectDeferDelete instead of **ObDereferenceObject** for any object, particularly Kernel Transaction Manager (KTM) objects, when the immediate deletion by the current thread of the object (by using **ObDereferenceObject**) might result in a deadlock.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

IoGetDeviceObjectPointer

ObDereferenceObjectDeferDelete

ZwClose

ZwMakeTemporaryObject

ObDereferenceObjectDeferDelete function

1/25/2019 • 2 minutes to read • Edit Online

The **ObDereferenceObjectDeferDelete** routine decrements the reference count for the given object, checks for object retention, and avoids deadlocks.

Syntax

```
void ObDereferenceObjectDeferDelete(
  PVOID Object
);
```

Parameters

`Object`

A pointer to the body of the object.

Return Value

None

Remarks

ObDereferenceObjectDeferDelete is similar to ObDereferenceObject except that, when the reference count of the object reaches zero, the object manager passes the object deletion request to a worker thread. Therefore, the deletion later occurs at IRQL = PASSIVE_LEVEL.

Use **ObDereferenceObjectDeferDelete** for any object when the immediate deletion by the current thread of the object (by using **ObDereferenceObject**) might result in a deadlock.

For example, such a deadlock can occur if **ObDereferenceObject** is used to dereference a Kernel Transaction Manager (KTM) object when a higher level driver on the driver stack is holding a lock.

To avoid such deadlocks, use **ObDereferenceObjectDeferDelete** instead of **ObDereferenceObject** to dereference KTM objects.

> **Note** For information about object permanence and attributes, see ObDereferenceObject.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows operating systems.
Target Platform	Universal
Header	wdm.h (include Fltkernel.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ObDereferenceObject

ObReferenceObject

ObDereferenceObjectDeferDeleteWithTag function

1/25/2019 • 2 minutes to read • Edit Online

The **ObDereferenceObjectDeferDeleteWithTag** routine decrements the reference count for the specified object, defers deletion of the object to avoid deadlocks, and writes a four-byte tag value to the object to support object reference tracing.

Syntax

```
void ObDereferenceObjectDeferDeleteWithTag(
  PVOID Object,
  ULONG Tag
);
```

Parameters

`Object`

A pointer to the object. The caller obtains this pointer either when it creates the object, or from a previous call to the ObReferenceObjectByHandleWithTag routine after it opens the object.

`Tag`

Specifies a four-byte, custom tag value. For more information, see the following Remarks section.

Return Value

None

Remarks

ObDereferenceObjectDeferDeleteWithTag is similar to ObDereferenceObjectWithTag except that, when the reference count of the object reaches zero, **ObDereferenceObjectDeferDeleteWithTag** passes the object deletion request to a worker thread. The worker thread, which runs at IRQL = PASSIVE_LEVEL, later deletes the object.

If the immediate deletion of an object by the current thread might cause a deadlock, do not call **ObDereferenceObjectWithTag** to dereference the object. Instead, call **ObDereferenceObjectDeferDeleteWithTag** to dereference the object.

For example, such a deadlock can occur if **ObDereferenceObjectWithTag** is used to dereference a Kernel Transaction Manager (KTM) object when a higher-level driver on the driver stack is holding a lock.

For more information about object permanence and object attributes, see ObDereferenceObjectWithTag. For more information about object references, see Life Cycle of an Object.

The ObDereferenceObjectDeferDelete routine is similar to **ObDereferenceObjectDeferDeleteWithTag**, except that it does not enable the caller to write a custom tag to an object. In Windows 7 and later versions of Windows, **ObDereferenceObjectDeferDelete** always writes a default tag value ('tIfD') to the object. A call to **ObDereferenceObjectDeferDelete** has the same effect as a call to **ObDereferenceObjectDeferDeleteWithTag** that specifies *Tag* = 'tIfD'.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger extension. In Windows 7, the !obtrace extension is enhanced to display object reference tags, if object reference tracing is enabled. By default, object reference tracing is turned off. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information, see Object Reference Tracing with Tags.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ObDereferenceObjectDeferDelete

ObDereferenceObjectWithTag

ObReferenceObjectByHandleWithTag

ObDereferenceObjectWithTag macro

1/25/2019 • 2 minutes to read • Edit Online

The **ObDereferenceObjectWithTag** routine decrements the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.

Syntax

```
void ObDereferenceObjectWithTag(
   a,
   t
);
```

Parameters

`a`

A pointer to the object. The caller obtains this pointer either when it creates the object, or from a previous call to the ObReferenceObjectByHandleWithTag routine after it opens the object.

`t`

Specifies a four-byte, custom tag value. For more information, see the following Remarks section.

Return Value

ObDereferenceObjectWithTag returns a value that is reserved for system use. Drivers must treat this value as VOID.

Remarks

A kernel-mode driver calls this routine to decrement the reference count of an object by one. If the object was created as *temporary* (that is, the OBJ_PERMANENT flag was not specified when the object was created), and the reference count reaches zero, the object manager deletes the object.

When the reference count for an object reaches zero, a kernel-mode component can delete the object. However, a driver can delete only those objects that it created, and a driver should never attempt to delete any object that it did not create.

An object is *permanent* if it was created with the OBJ_PERMANENT object attribute flag specified. (For more information about object attributes, see InitializeObjectAttributes.) A permanent object is created with an initial reference count of one, so the object is not deleted when the driver removes its last reference to the object.

To prepare a permanent object to be deleted, a driver can call the ZwMakeTemporaryObject routine to make the object temporary. A driver should only delete a permanent object that it created. Use the following steps to delete a permanent object:

1. Call **ObDereferenceObjectWithTag**.
2. Get the handle to the object. If necessary, call the appropriate **ZwOpen*Xxx*** or **ZwCreate*Xxx*** routine to get the object handle.

3. Call **ZwMakeTemporaryObject** with the handle obtained in step 2.

4. Call ZwClose with the handle obtained in step 2.

If the immediate deletion of an object by the current thread might cause a deadlock, do not call **ObDereferenceObjectWithTag** to dereference the object. Instead, call the ObDereferenceObjectDeferDeleteWithTag routine to dereference the object.

For example, such a deadlock can occur if **ObDereferenceObjectWithTag** is used to dereference a Kernel Transaction Manager (KTM) object when a higher-level driver on the driver stack is holding a lock.

For more information about object references, see Life Cycle of an Object.

The ObDereferenceObject routine is similar to **ObDereferenceObjectWithTag**, except that it does not enable the caller to write a custom tag to an object. In Windows 7 and later versions of Windows, **ObDereferenceObject** always writes a default tag value ('tIfD') to the object. A call to **ObDereferenceObject** has the same effect as a call to **ObDereferenceObjectWithTag** that specifies *Tag* = 'tIfD'.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger extension. In Windows 7, the **!obtrace** extension is enhanced to display object reference tags, if object reference tracing is enabled. By default, object reference tracing is turned off. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information, see Object Reference Tracing with Tags.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

IoGetDeviceObjectPointer

ObDereferenceObject

ObDereferenceObjectDeferDeleteWithTag

ObReferenceObjectByHandleWithTag

ZwClose

ZwMakeTemporaryObject

ObfReferenceObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObject** routine increments the reference count to the given object.

Syntax

```
LONG_PTR ObfReferenceObject(
  PVOID Object
);
```

Parameters

`Object`

Pointer to the object. The caller obtained this parameter either when it created the object or from a preceding call to ObReferenceObjectByHandle after it opened the object.

Return Value

ObReferenceObject returns a value that is reserved for system use. Drivers must treat this value as VOID.

Remarks

ObReferenceObject simply increments the pointer reference count for an object, without making any access checks on the given object, as ObReferenceObjectByHandle and ObReferenceObjectByPointer do.

ObReferenceObject prevents deletion of the object at least until the driver subsequently calls its reciprocal, **ObDereferenceObject**, or closes the given object. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

When the reference count for an object reaches zero, a kernel-mode component can remove the object from the system. However, a driver can remove only those objects that it created, and a driver should never attempt to remove any object that it did not create.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<=DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ObDereferenceObject

ObReferenceObjectByHandle

ObReferenceObjectByPointer

ZwClose

ObGetObjectSecurity function

1/25/2019 • 2 minutes to read • Edit Online

The **ObGetObjectSecurity** routine gets the security descriptor for a given object.

Syntax

```
NTSTATUS ObGetObjectSecurity(
  PVOID                Object,
  PSECURITY_DESCRIPTOR *SecurityDescriptor,
  PBOOLEAN             MemoryAllocated
);
```

Parameters

`Object`

Pointer to the object.

`SecurityDescriptor`

Pointer to a caller-supplied variable that this routine sets to the address of a buffer containing the SECURITY_DESCRIPTOR for the given object. If the given object has no security descriptor, this variable is set to **NULL** on return from **ObGetObjectSecurity**.

`MemoryAllocated`

Pointer to a caller-supplied variable that this routine sets to **TRUE** if it allocated a buffer to contain the security descriptor.

Return Value

ObGetObjectSecurity either returns STATUS_SUCCESS or an error status, such as STATUS_INSUFFICIENT_RESOURCES if it could not allocate enough memory to return the requested information.

Remarks

A successful call to **ObGetObjectSecurity** either returns a self-relative security descriptor in the buffer at **SecurityDescriptor* or it returns **NULL** at **SecurityDescriptor* if the given object has no security descriptor. For example, any unnamed object, such as an event object, has no security descriptor.

If **ObGetObjectSecurity** returns STATUS_SUCCESS, the caller must save the value returned at *MemoryAllocated*. Such a caller must pass *MemoryAllocated* in a reciprocal call to ObReleaseObjectSecurity eventually, thereby restoring the reference count on the security descriptor to its original value and releasing the buffer, if any, that was allocated by **ObGetObjectSecurity**.

ObGetObjectSecurity should only be called at IRQL Level = PASSIVE_LEVEL with APCs enabled, otherwise deadlocks or crashes may occur.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlApcLte, HwStorPortProhibitedDDIs

See Also

ObReferenceObjectByHandle

ObReleaseObjectSecurity

SECURITY_DESCRIPTOR

ObReferenceObject macro

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObject** routine increments the reference count to the given object.

Syntax

```
void ObReferenceObject(
   Object
);
```

Parameters

`Object`

Pointer to the object. The caller obtained this parameter either when it created the object or from a preceding call to ObReferenceObjectByHandle after it opened the object.

Return Value

ObReferenceObject returns a value that is reserved for system use. Drivers must treat this value as VOID.

Remarks

ObReferenceObject simply increments the pointer reference count for an object, without making any access checks on the given object, as ObReferenceObjectByHandle and ObReferenceObjectByPointer do.

ObReferenceObject prevents deletion of the object at least until the driver subsequently calls its reciprocal, **ObDereferenceObject**, or closes the given object. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

When the reference count for an object reaches zero, a kernel-mode component can remove the object from the system. However, a driver can remove only those objects that it created, and a driver should never attempt to remove any object that it did not create.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"<=DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ObDereferenceObject

ObReferenceObjectByHandle

ObReferenceObjectByPointer

ZwClose

ObReferenceObjectByHandle function

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObjectByHandle** routine provides access validation on the object handle, and, if access can be granted, returns the corresponding pointer to the object's body.

Syntax

```
NTSTATUS ObReferenceObjectByHandle(
  HANDLE                     Handle,
  ACCESS_MASK                DesiredAccess,
  POBJECT_TYPE               ObjectType,
  KPROCESSOR_MODE            AccessMode,
  PVOID                      *Object,
  POBJECT_HANDLE_INFORMATION HandleInformation
);
```

Parameters

`Handle`

Specifies an open handle for an object.

`DesiredAccess`

Specifies the requested types of access to the object. The interpretation of this field is dependent on the object type. Do not use any generic access rights. For more information, see ACCESS_MASK.

`ObjectType`

Pointer to the object type. *ObjectType* can be ***ExEventObjectType**, ***ExSemaphoreObjectType**, ***IoFileObjectType**, ***PsProcessType**, ***PsThreadType**, ***SeTokenObjectType**, ***TmEnlistmentObjectType**, ***TmResourceManagerObjectType**, ***TmTransactionManagerObjectType**, or ***TmTransactionObjectType**.

If *ObjectType* is not **NULL**, the operating system verifies that the supplied object type matches the object type of the object that *Handle* specifies.

`AccessMode`

Specifies the access mode to use for the access check. It must be either **UserMode** or **KernelMode**. Drivers should always specify **UserMode** for handles they receive from user address space.

`Object`

Pointer to a variable that receives a pointer to the object's body. The following table contains the pointer types.

OBJECTTYPE PARAMETER	OBJECT POINTER TYPE
***ExEventObjectType**	PKEVENT
***ExSemaphoreObjectType**	PKSEMAPHORE
***IoFileObjectType**	PFILE_OBJECT

OBJECTTYPE PARAMETER	OBJECT POINTER TYPE
***PsProcessType**	PEPROCESS or PKPROCESS
***PsThreadType**	PETHREAD or PKTHREAD
***SeTokenObjectType**	PACCESS_TOKEN
***TmEnlistmentObjectType**	PKENLISTMENT
***TmResourceManagerObjectType**	PKRESOURCEMANAGER
***TmTransactionManagerObjectType**	PKTM
***TmTransactionObjectType**	PKTRANSACTION

The structures that the pointer types reference are opaque, and drivers cannot access the structure members. Because the structures are opaque, PEPROCESS is equivalent to PKPROCESS, and PETHREAD is equivalent to PKTHREAD.

`HandleInformation`

Drivers set this to **NULL**.

Return Value

ObReferenceObjectByHandle returns STATUS_SUCCESS if the call is successful. Possible return values include the following error codes:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The *ObjectType* parameter specifies the wrong object type for the object that is identified by the *Handle* parameter.
STATUS_ACCESS_DENIED	The caller cannot be granted the requested access rights to the object.
STATUS_INVALID_HANDLE	The *Handle* parameter is not a valid object handle.

Remarks

A pointer to the object body is retrieved from the object table entry and returned to the caller by means of the *Object* parameter.

If *AccessMode* is **UserMode**, the requested access is compared to the granted access for the object. If *AccessMode* is **KernelMode**, the handle should originate in the kernel address space.

Starting with Windows 7, if *AccessMode* is **KernelMode** and handle is received from user address space, Driver Verifier issues bugcheck C4, subcode F6.

If the call succeeds, a pointer to the object body is returned to the caller and the pointer reference count is incremented. Incrementing this count prevents the object from being deleted while the pointer is being referenced. The caller must decrement the reference count with ObDereferenceObject as soon as it is done with the object.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlObPassive, TargetRelationNeedsRef, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

ObDereferenceObject

ObReferenceObject

ObReferenceObjectByPointer

ObReferenceObjectByHandleWithTag function

1/25/2019 • 3 minutes to read • Edit Online

The **ObReferenceObjectByHandleWithTag** routine increments the reference count of the object that is identified by the specified handle, and writes a four-byte tag value to the object to support object reference tracing.

Syntax

```
NTSTATUS ObReferenceObjectByHandleWithTag(
  HANDLE                     Handle,
  ACCESS_MASK                DesiredAccess,
  POBJECT_TYPE               ObjectType,
  KPROCESSOR_MODE            AccessMode,
  ULONG                      Tag,
  PVOID                      *Object,
  POBJECT_HANDLE_INFORMATION HandleInformation
);
```

Parameters

`Handle`

Specifies an open handle for an object.

`DesiredAccess`

Specifies the types of access to the object that the caller requests. This parameter is a bitmask of type ACCESS_MASK. The interpretation of this field depends on the object type. Do not use any generic access rights.

`ObjectType`

A pointer to an opaque structure that specifies the object type. This parameter points to an OBJECT_TYPE structure. Set *ObjectType* to **NULL** or to one of the following pointer values, which are declared in the Wdm.h header file: ***ExEventObjectType**, ***ExSemaphoreObjectType**, ***IoFileObjectType**, ***PsProcessType**, ***PsThreadType**, ***SeTokenObjectType**, ***TmEnlistmentObjectType**, ***TmResourceManagerObjectType**, ***TmTransactionManagerObjectType**, or ***TmTransactionObjectType**. If *ObjectType* is not **NULL**, the routine verifies that the supplied object type matches the object type of the object that the *Handle* parameter specifies.

`AccessMode`

Specifies the access mode to use for the access check. It must be either **UserMode** or **KernelMode**. Drivers should always specify **UserMode** for handles they receive from user address space.

`Tag`

Specifies a four-byte, custom tag value. For more information, see the following Remarks section.

`Object`

A pointer to a variable into which the routine writes a pointer to the object. The following table lists the *Object* pointer types that are designated by the possible *ObjectType* parameter values.

ObjectType Parameter	*Object* Pointer Type
***ExEventObjectType**	PKEVENT
***ExSemaphoreObjectType**	PKSEMAPHORE
***IoFileObjectType**	PFILE_OBJECT
***PsProcessType**	PEPROCESS or PKPROCESS
***PsThreadType**	PETHREAD or PKTHREAD
***SeTokenObjectType**	PACCESS_TOKEN
***TmEnlistmentObjectType**	PKENLISTMENT
***TmResourceManagerObjectType**	PKRESOURCEMANAGER
***TmTransactionManagerObjectType**	PKTM
***TmTransactionObjectType**	PKTRANSACTION

The structures that the pointer types reference are opaque, and drivers cannot access the structure members. Because the structures are opaque, PEPROCESS is equivalent to PKPROCESS, and PETHREAD is equivalent to PKTHREAD.

`HandleInformation`

Drivers set this parameter to **NULL**.

Return Value

ObReferenceObjectByHandleWithTag returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

Return code	Description
STATUS_OBJECT_TYPE_MISMATCH	The *ObjectType* parameter specifies the wrong object type for the object that is identified by the *Handle* parameter.
STATUS_ACCESS_DENIED	The caller does not have the required access rights to the object.
STATUS_INVALID_HANDLE	The specified handle is not valid.

Remarks

This routine does access validation of the specified object handle. If access can be granted, the routine increments the object reference count and provides an object pointer to the caller. This increment prevents the object from being deleted while the caller uses the object. When the object is no longer needed, the caller should decrement the reference count by calling the ObDereferenceObjectWithTag or ObDereferenceObjectDeferDeleteWithTag routine.

For more information about object references, see Life Cycle of an Object.

ObReferenceObjectByHandleWithTag does not close or invalidate the object handle that is specified by the *Handle* parameter. When the handle is no longer needed, the caller can close the handle by calling the ZwClose routine.

If the *AccessMode* parameter value is **KernelMode**, the requested access is always allowed. If *AccessMode* is **UserMode**, the requested access is compared to the access rights that the caller has to the object. Only highest-level drivers can safely specify the **UserMode** value for the *AccessMode* parameter.

Starting with Windows 7, if *AccessMode* is **KernelMode** and handle is received from user address space, Driver Verifier issues bugcheck C4, subcode F6.

The ObReferenceObjectByHandle routine is similar to **ObReferenceObjectByHandleWithTag**, except that it does not enable the caller to write a custom tag to an object. In Windows 7 and later versions of Windows, **ObReferenceObjectByHandle** always writes a default tag value ('tIfD') to the object. A call to **ObReferenceObjectByHandle** has the same effect as a call to **ObReferenceObjectByHandleWithTag** that specifies *Tag* = 'tIfD'.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger extension. The **!obtrace** extension is enhanced to display object reference tags, if object reference tracing is enabled. By default, object reference tracing is turned off. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information, see Object Reference Tracing with Tags.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

OBJECT_TYPE

ObDereferenceObjectDeferDeleteWithTag

ObDereferenceObjectWithTag

ObReferenceObjectByHandle

ZwClose

ObReferenceObjectByPointer function

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObjectByPointer** routine increments the pointer reference count for a given object.

Syntax

```
NTSTATUS ObReferenceObjectByPointer(
  PVOID           Object,
  ACCESS_MASK     DesiredAccess,
  POBJECT_TYPE    ObjectType,
  KPROCESSOR_MODE AccessMode
);
```

Parameters

`Object`

Pointer to the object's body.

`DesiredAccess`

Specifies a mask representing the requested access to the object.

`ObjectType`

Pointer to the object type. *ObjectType* can be ***ExEventObjectType***, **ExSemaphoreObjectType**, ***IoFileObjectType***, **PsProcessType**, ***PsThreadType***, **SeTokenObjectType**, ***TmEnlistmentObjectType***, **TmResourceManagerObjectType**, ***TmTransactionManagerObjectType***, or **TmTransactionObjectType**.

> **Note** The **SeTokenObjectType** object type is supported in Windows XP and later versions of Windows.

This parameter can also be **NULL** if *AccessMode* is **KernelMode**.

`AccessMode`

Indicates the access mode to use for the access check. It must be either **UserMode** or **KernelMode**. Lower-level drivers should specify **KernelMode**.

Return Value

ObReferenceObjectByPointer returns an NTSTATUS value. Possible return values include:

Remarks

Calling this routine prevents the object from being deleted, possibly by another component's call to ObDereferenceObject or ZwClose. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	TargetRelationNeedsRef, HwStorPortProhibitedDDIs

See Also

ObDereferenceObject

ObReferenceObject

ObReferenceObjectByHandle

ZwClose

ObReferenceObjectByPointerWithTag function

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObjectByPointerWithTag** routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.

Syntax

```
NTSTATUS ObReferenceObjectByPointerWithTag(
  PVOID           Object,
  ACCESS_MASK     DesiredAccess,
  POBJECT_TYPE    ObjectType,
  KPROCESSOR_MODE AccessMode,
  ULONG           Tag
);
```

Parameters

`Object`

A pointer to the object. The caller obtains this pointer either when it creates the object, or from a previous call to the ObReferenceObjectByHandleWithTag routine after it opens the object.

`DesiredAccess`

Specifies the types of access to the object that the caller requests. This parameter is a bitmask of type ACCESS_MASK. The interpretation of this field depends on the object type. Do not use any generic access rights.

`ObjectType`

A pointer to an opaque structure that specifies the object type. This parameter points to an OBJECT_TYPE structure. Set *ObjectType* to **NULL** or to one of the following pointer values, which are declared in the Wdm.h header file: ***ExEventObjectType***, **ExSemaphoreObjectType**, ***IoFileObjectType***, **PsProcessType**, ***PsThreadType***, **SeTokenObjectType**, ***TmEnlistmentObjectType***, **TmResourceManagerObjectType**, ***TmTransactionManagerObjectType***, or **TmTransactionObjectType**. This parameter can be **NULL** if *AccessMode* is **KernelMode**. If *ObjectType* is not **NULL**, the routine verifies that the supplied object type matches the object type of the object that the *Handle* parameter specifies.

`AccessMode`

Indicates the access mode to use for the access check. Set this parameter to one of the following **MODE** enumeration values:

- **UserMode**
- **KernelMode**

Lower-level drivers should specify **KernelMode**.

`Tag`

Specifies a four-byte, custom tag value. For more information, see the following Remarks section.

Return Value

ObReferenceObjectByPointerWithTag returns STATUS_SUCCESS if the call is successful. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STA TUS _OB JEC T_T YPE _MI SM ATC H	The *ObjectType* parameter specifies the wrong object type for the object that the *Object* parameter points to, or *ObjectType* is **NULL** but *AccessMode* is **UserMode**.

Remarks

This routine does access validation of the specified object. If access can be granted, the routine increments the object reference count. This increment prevents the object from being deleted while the caller uses the object. When the object is no longer needed, the caller should decrement the reference count by calling the ObDereferenceObjectWithTag or ObDereferenceObjectDeferDeleteWithTag routine.

For more information about object references, see Life Cycle of an Object.

The ObReferenceObjectByPointer routine is similar to **ObReferenceObjectByPointerWithTag**, except that it does not enable the caller to write a custom tag to an object. In Windows 7 and later versions of Windows, **ObReferenceObjectByPointer** always writes a default tag value ('tIfD') to the object. A call to **ObReferenceObjectByPointer** has the same effect as a call to **ObReferenceObjectByPointerWithTag** that specifies *Tag* = 'tIfD'.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger extension. In Windows 7, the !obtrace extension is enhanced to display object reference tags, if object reference tracing is enabled. By default, object reference tracing is turned off. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information, see Object Reference Tracing with Tags.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

OBJECT_TYPE

ObDereferenceObjectDeferDeleteWithTag

ObDereferenceObjectWithTag

ObReferenceObjectByPointer

ZwClose

ObReferenceObjectWithTag macro

1/25/2019 • 2 minutes to read • Edit Online

The **ObReferenceObjectWithTag** routine increments the reference count of the specified object, and writes a four-byte tag value to the object to support object reference tracing.

Syntax

```
void ObReferenceObjectWithTag(
   Object,
   Tag
);
```

Parameters

`Object`

A pointer to the object. The caller obtains this pointer either when it creates the object, or from a previous call to the ObReferenceObjectByHandleWithTag routine after it opens the object.

`Tag`

Specifies a four-byte, custom tag value. For more information, see the following Remarks section.

Return Value

ObReferenceObjectWithTag returns a value that is reserved for system use. Drivers must treat this value as VOID.

Remarks

ObReferenceObjectWithTag simply increments the pointer reference count for an object, without making any access checks on the specified object. In contrast, the ObReferenceObjectByHandleWithTag and ObReferenceObjectByPointerWithTag routines verify that the caller has the required access rights to the object and fail if the caller does not have these rights.

A **ObReferenceObjectWithTag** call prevents deletion of the specified object at least until the driver either calls the ObDereferenceObjectWithTag routine, or closes the object. After the object is no longer needed, the driver must call **ObDereferenceObjectWithTag** to remove its reference to the object.

When the reference count for an object reaches zero, a kernel-mode component can delete the object. However, a driver can delete only those objects that it created, and a driver should never attempt to delete any object that it did not create.

For more information about object references, see Life Cycle of an Object.

The ObReferenceObject routine is similar to **ObReferenceObjectWithTag**, except that it does not enable the caller to write a custom tag to an object. In Windows 7 and later versions of Windows, **ObReferenceObject** always writes a default tag value ('tIfD') to the object. A call to **ObReferenceObject** has the same effect as a call to **ObReferenceObjectWithTag** that specifies *Tag* = 'tIfD'.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger

extension. In Windows 7, the !obtrace extension is enhanced to display object reference tags, if object reference tracing is enabled. By default, object reference tracing is turned off. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information, see Object Reference Tracing with Tags.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

ObDereferenceObjectWithTag

ObReferenceObject

ObReferenceObjectByHandleWithTag

ObReferenceObjectByPointerWithTag

ObRegisterCallbacks function

1/25/2019 • 2 minutes to read • Edit Online

The **ObRegisterCallbacks** routine registers a list of callback routines for thread, process, and desktop handle operations.

Syntax

```
NTSTATUS ObRegisterCallbacks(
  POB_CALLBACK_REGISTRATION CallbackRegistration,
  PVOID                     *RegistrationHandle
);
```

Parameters

`CallbackRegistration`

A pointer to an OB_CALLBACK_REGISTRATION structure that specifies the list of callback routines and other registration information.

`RegistrationHandle`

A pointer to a variable that receives a value that identifies the set of registered callback routines. The caller passes this value to the ObUnRegisterCallbacks routine to unregister the set of callbacks.

Return Value

ObRegisterCallbacks returns an NTSTATUS value. This routine might return one of the following values:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The specified callback routines are registered with the system.
STA TUS _FLT _IN STA NCE _AL TIT UDE _CO LLIS ION	The calling driver or another driver has already registered callback routines for the altitude that *CallBackRegistration*->**Altitude** specifies. For more information about this altitude, see the description of the **Altitude** member in OB_CALLBACK_REGISTRATION.

STATUS_INVALID_PARAMETER	One or more of the parameters that were specified in the registration was invalid. **ObRegisterCallbacks** might return this error, for example, if an invalid value for *CallBackRegistration*->**Version** is specified or if registration is attempted for object types that do not support callback routines.
STATUS_ACCESS_DENIED	The callback routines do not reside in a signed kernel binary image.
STATUS_INSUFFICIENT_RESOURCES	An attempt to allocate memory failed.

Remarks

A driver must unregister all callback routines before it unloads. You can unregister the callback routine by calling the **ObUnRegisterCallbacks** routine.

Requirements

Minimum supported client	Available starting with Windows Vista with Service Pack 1 (SP1) and Windows Server 2008.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

OB_CALLBACK_REGISTRATION

ObUnRegisterCallbacks

ObReleaseObjectSecurity function

1/25/2019 • 2 minutes to read • Edit Online

The **ObReleaseObjectSecurity** routine is the reciprocal to **ObGetObjectSecurity**.

Syntax

```
void ObReleaseObjectSecurity(
  PSECURITY_DESCRIPTOR SecurityDescriptor,
  BOOLEAN              MemoryAllocated
);
```

Parameters

`SecurityDescriptor`

Pointer to the buffered SECURITY_DESCRIPTOR to be released. The caller obtained this parameter from **ObGetObjectSecurity**

`MemoryAllocated`

Specifies the value also obtained from **ObGetObjectSecurity**.

Return Value

None

Remarks

After a successful call to ObGetObjectSecurity, a driver must call **ObReleaseObjectSecurity** eventually.

ObReleaseObjectSecurity releases any resources that were allocated by **ObGetObjectSecurity**. It also decrements the reference count on the given security descriptor.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

DDI compliance rules	IrqlApcLte, HwStorPortProhibitedDDIs

See Also

ObGetObjectSecurity

SECURITY_DESCRIPTOR

ObUnRegisterCallbacks function

1/25/2019 • 2 minutes to read • Edit Online

The **ObUnRegisterCallbacks** routine unregisters a set of callback routines that were registered with the ObRegisterCallbacks routine.

Syntax

```
void ObUnRegisterCallbacks(
  PVOID RegistrationHandle
);
```

Parameters

`RegistrationHandle`

A value that identifies the set of callback routines to unregister. The ObRegisterCallbacks routine provides this value when it originally registered the callback routines.

Return Value

None

Remarks

A driver that calls the **ObRegisterCallbacks** routine must call the **ObUnRegisterCallbacks** routine before the driver is unloaded.

Requirements

Minimum supported client	Available in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

ObRegisterCallbacks

PALLOCATE_ADAPTER_CHANNEL callback function

1/25/2019 • 2 minutes to read • Edit Online

The **AllocateAdapterChannel** routine prepares the system for a DMA operation on behalf of the target device object, and then calls the driver-supplied AdapterControl routine to carry out the DMA operation.

Syntax

```
PALLOCATE_ADAPTER_CHANNEL PallocateAdapterChannel;

NTSTATUS PallocateAdapterChannel(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  ULONG NumberOfMapRegisters,
  PDRIVER_CONTROL ExecutionRoutine,
  PVOID Context
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`DeviceObject`

Pointer to the device object that represents the target device for a requested DMA operation.

`NumberOfMapRegisters`

Specifies the number of map registers to be used in the transfer. This value is the lesser of the number of map registers needed to satisfy the current transfer request, and the number of available map registers returned by IoGetDmaAdapter.

`ExecutionRoutine`

Pointer to a driver-supplied AdapterControl routine. The routine is called when the system DMA controller or bus-master adapter becomes available.

`Context`

Pointer to the driver-determined context to be passed to the AdapterControl routine.

Return Value

This routine can return one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION

STA TUS _SU CCE SS	The adapter channel has been allocated. The system will call the *AdapterControl* routine once the DMA operation can begin.
STA TUS _IN SUF FICI ENT _RE SO URC ES	The *NumberOfMapRegisters* is larger than the value returned by **IoGetDmaAdapter**. The *AdapterControl* routine will not be called.

Remarks

AllocateAdapterChannel is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

A driver calls the **AllocateAdapterControl** routine to register an AdapterControl routine that performs a DMA operation for the driver. The *AdapterControl* routine carries out a DMA operation using either the system DMA controller or a bus-master adapter.

If the DMA operation can be performed immediately, the system immediately calls *AdapterControl*. If the system DMA controller or bus-master adapter is currently in use, **AllocateAdapterChannel** queues the *AdapterControl* until the adapter becomes available. In either case, **AllocateAdapterChannel** returns STATUS_SUCCESS.

If the system lacks the resources to perform the DMA operation, **AllocateAdapterChannel** returns STATUS_INSUFFICIENT_RESOURCES. In that case, *AdapterControl* is not queued, and the driver should complete the current IRP with the appropriate error code (such as STATUS_INSUFFICIENT_RESOURCES).

This routine reserves exclusive access to a DMA controller channel and map registers for the one or more DMA operations that are required to satisfy the current IRP's transfer request for the specified device.

Only one DMA request can be queued for a device object at any one time. Therefore, the driver should not call **AllocateAdapterChannel** again for another DMA operation on the same device object until the AdapterControl routine has completed execution. In addition, a driver must not call **AllocateAdapterChannel** from within its *AdapterControl* routine.

The system passes the value of the **CurrentIrp** member of *DeviceObject* as the *Irp* parameter of *AdapterControl*. If **AllocateAdapterChannel** is called from a driver's StartIo routine, this is guaranteed to point to the IRP that *StartIo* was called to process. Otherwise, to use the *Irp* parameter of *AdapterControl*, the driver must set **CurrentIrp** to point to the current IRP before calling **AllocateAdapterChannel**.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, IrqlDispatch(storport)

See Also

DEVICE_OBJECT

DMA_ADAPTER

DMA_OPERATIONS

FlushAdapterBuffers

FreeAdapterChannel

FreeMapRegisters

IoGetDmaAdapter

MapTransfer

ReadDmaCounter

PALLOCATE_ADAPTER_CHANNEL_EX callback function

1/25/2019 • 4 minutes to read • Edit Online

The **AllocateAdapterChannelEx** routine allocates the resources that are needed to perform a DMA transfer, and then calls the driver-supplied AdapterControl routine to initiate the DMA transfer.

Syntax

```
PALLOCATE_ADAPTER_CHANNEL_EX PallocateAdapterChannelEx;

NTSTATUS PallocateAdapterChannelEx(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PVOID DmaTransferContext,
  ULONG NumberOfMapRegisters,
  ULONG Flags,
  PDRIVER_CONTROL ExecutionRoutine,
  PVOID ExecutionContext,
  PVOID *MapRegisterBase
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`DeviceObject`

A pointer to a DEVICE_OBJECT structure. This structure is the physical device object (PDO) that represents the target device for the requested DMA operation.

`DmaTransferContext`

A pointer to an initialized DMA transfer context. This context was initialized by a previous call to the InitializeDmaTransferContext routine. This context must be unique across all adapter allocation requests. To cancel a pending allocation request, the caller must supply the DMA transfer context for the request to the CancelAdapterChannel routine.

`NumberOfMapRegisters`

The number of map registers to use in the DMA transfer. The calling driver should set this value to the lesser of the number of map registers needed to satisfy the current transfer request, and the number of available map registers. The driver previously called the GetDmaTransferInfo routine to obtain the number of map registers needed for the transfer, and called the IoGetDmaAdapter routine to obtain the number of available map registers.

`Flags`

The adapter channel allocation flags. The following flag is supported.

FLAG	MEANING
DM A_S YN CH RO NO US_ CAL LBA CK	The **AllocateAdapterChannelEx** routine is called synchronously. If this flag is set, and the required DMA resources are not immediately available, the call fails and returns STATUS_INSUFFICIENT_RESOURCES.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, the *ExecutionRoutine* parameter is optional and can be NULL. For more information about this flag, see the Remarks section.

`ExecutionRoutine`

A pointer to the driver-supplied AdapterControl routine that initiates the DMA transfer for the driver. The I/O manager calls the *AdapterControl* routine after the required resources are allocated for the adapter object. After the *AdapterControl* routine returns, the I/O manager automatically frees the adapter object. The I/O manager might additionally free the resources that were allocated for this object, depending on the value returned by this routine.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, the *ExecutionRoutine* is optional and can be **NULL**. In this case, the caller can use the resources allocated by **AllocateAdapterChannelEx**, and later free these resources by calling the FreeAdapterObject routine. For more information, see the Remarks section.

`ExecutionContext`

The driver-determined, adapter-control context. This context is passed to the AdapterControl routine as the *Context* parameter.

`*MapRegisterBase`

A pointer to a variable into which the routine writes a handle to the allocated map registers. The caller can supply this handle as a parameter to the FlushAdapterBuffersEx, FlushAdapterBuffers, FreeMapRegisters, or MapTransferEx routine.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, *MapRegisterBase* must be a valid, non-**NULL** pointer. If the *ExecutionRoutine* parameter is non-**NULL**, *MapRegisterBase* must be **NULL**. The call fails if *MapRegisterBase* is non-**NULL** and the **DMA_SYNCHRONOUS_CALLBACK** flag is not set, or if *MapRegisterBase* and *ExecutionRoutine* are both **NULL**.

Return Value

AllocateAdapterChannelEx returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETERS	The routine failed due to invalid parameter values passed by the caller.
STATUS_INSUFFICIENT_RESOURCES	The routine failed to allocate resources required for the DMA transfer.

Remarks

AllocateAdapterChannelEx
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

AllocateAdapterChannelEx allocates the resources that are required to perform a DMA operation. These resources include DMA channels and map registers. After all required resources are allocated for use by the DMA adapter, **AllocateAdapterChannelEx** calls the caller-supplied AdapterControl routine to initiate the DMA operation.

By default, **AllocateAdapterChannelEx** returns asynchronously, without waiting for the requested resource allocation to complete. After this return, the caller can, if necessary, cancel the pending allocation request by calling the CancelAdapterChannel routine.

If the calling driver sets the **DMA_SYNCHRONOUS_CALLBACK** flag, the **AllocateAdapterChannelEx** routine behaves as follows:

- If the requested DMA resources are not immediately available, **AllocateAdapterChannelEx** does not wait for resources and does not call the AdapterControl routine. Instead, **AllocateAdapterChannelEx** fails and returns STATUS_INSUFFICIENT_RESOURCES.
- The driver is not required to supply an AdapterControl routine if the **DMA_SYNCHRONOUS_CALLBACK** flag is set.
- If the driver supplies an AdapterControl routine, the **DMA_SYNCHRONOUS_CALLBACK** flag indicates that this routine is to be called in the context of the calling thread, before **AllocateAdapterChannelEx** returns.
- If the driver does not supply an AdapterControl routine, the driver can use the allocated resources after **AllocateAdapterChannelEx** returns. In this case, the driver must call FreeAdapterObject after it finishes using the allocated resources.

AllocateAdapterChannelEx is an extended version of the AllocateAdapterChannel routine. The following features are available only in the extended version:

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL

See Also

AdapterControl

AllocateAdapterChannel

CancelAdapterChannel

DEVICE_OBJECT

DMA_ADAPTER

DMA_OPERATIONS

FlushAdapterBuffers

FlushAdapterBuffersEx

FreeAdapterObject

FreeMapRegisters

GetDmaTransferInfo

InitializeDmaTransferContext

IoGetDmaAdapter

MapTransferEx

PALLOCATE_COMMON_BUFFER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **AllocateCommonBuffer** routine allocates memory and maps it so that it is simultaneously accessible from both the processor and a device for DMA operations.

Syntax

```
PALLOCATE_COMMON_BUFFER PallocateCommonBuffer;

PVOID PallocateCommonBuffer(
  PDMA_ADAPTER DmaAdapter,
  ULONG Length,
  PPHYSICAL_ADDRESS LogicalAddress,
  BOOLEAN CacheEnabled
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`Length`

Specifies the number of bytes of memory to allocate.

`LogicalAddress`

Pointer to a variable that receives the logical address the device can use to access the allocated buffer. Use this address rather than calling MmGetPhysicalAddress because the system can take into account any platform-specific memory restrictions.

`CacheEnabled`

Specifies whether the allocated memory can be cached.

This parameter is ignored. The operating system determines whether to enable cached memory in the common buffer that is to be allocated. That decision is based on the processor architecture and device bus.

On computers with x86-based, x64-based, and Itanium-based processors, cached memory is enabled. It is assumed that all DMA operations performed by a device are coherent with the relevant CPU caches, which might be caching that memory. If your driver needs to disable caching, call AllocateCommonBufferEx instead.

On computers with ARM or ARM 64-based processors, the operating system does not automatically enable cached memory for all devices. The system relies on the **ACPI_CCA** method for each device to determine whether the device is cache-coherent.

Return Value

AllocateCommonBuffer returns the base virtual address of the allocated range. If the buffer cannot be allocated, it returns **NULL**.

Remarks

AllocateCommonBuffer is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a **DMA_OPERATIONS** structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

AllocateCommonBuffer supports DMA in which the device and the processor continuously communicate through system memory, as in a control structure for a bus-master DMA device.

AllocateCommonBuffer also supports subordinate devices whose drivers use a system DMA controller's autoinitialize mode.

AllocateCommonBuffer does the following:

- Allocates memory that can be reached from both the processor and the device. This memory appears contiguous to the device.
- Allocates map registers to map the buffer, if required by the system.
- Sets up a translation for the device, including loading map registers if necessary.

To use resident system memory economically, drivers should allocate as few of these buffers per device as possible. **AllocateCommonBuffer** allocates at least a page of memory, regardless of the requested *Length*. After a successful allocation requesting fewer than PAGE_SIZE bytes, the caller can access only the requested *Length*. After a successful allocation requesting more than an integral multiple of PAGE_SIZE bytes, any remaining bytes on the last allocated page are inaccessible to the caller.

If a driver needs several pages of common buffer space, but the pages need not be contiguous, the driver should make several one-page requests to **AllocateCommonBuffer** instead of one large request. This approach conserves contiguous memory.

Drivers typically call **AllocateCommonBuffer** as part of device start-up, during their response to a PnP IRP_MN_START_DEVICE request. After startup, it is possible that only one-page requests will succeed, if any.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlDispatch

See Also

DMA_ADAPTER

DMA_OPERATIONS

FreeCommonBuffer

IoGetDmaAdapter

PALLOCATE_COMMON_BUFFER_EX callback function

1/25/2019 • 3 minutes to read • Edit Online

The **AllocateCommonBufferEx** routine allocates memory for a common buffer and maps this memory so that it can be accessed both by the processor and by a device that performs DMA operations.

Syntax

```
PALLOCATE_COMMON_BUFFER_EX PallocateCommonBufferEx;

PVOID PallocateCommonBufferEx(
  PDMA_ADAPTER DmaAdapter,
  PPHYSICAL_ADDRESS MaximumAddress,
  ULONG Length,
  PPHYSICAL_ADDRESS LogicalAddress,
  BOOLEAN CacheEnabled,
  NODE_REQUIREMENT PreferredNode
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`MaximumAddress`

A pointer to a variable that contains the maximum logical address for the common buffer. This parameter indicates that the buffer should be allocated from memory below this address. This parameter is optional and can be specified as NULL to indicate that there is no maximum address.

`Length`

The size, in bytes, of the common buffer that is to be allocated for the DMA operation.

`LogicalAddress`

A pointer to a variable into which this routine writes the logical address that the device can use to access the common buffer. The DMA device should use this logical address instead of the physical address that is returned by a routine such as MmGetPhysicalAddress.

`CacheEnabled`

Whether the routine must enable or disable cached memory in the common buffer that is to be allocated. If TRUE, caching is enabled. If FALSE, it is disabled. If the hardware platform does not enforce cache coherency for DMA operations, then pass FALSE. For information about this parameter on ARM or ARM 64-based processors target computers, see Remarks.

`PreferredNode`

The preferred NUMA node from which the memory is to be allocated. If N is the number of NUMA nodes in a multiprocessor system, *PreferredNode* is a number in the range 0 to N–1. For a one-processor system or a non-NUMA multiprocessor system, set *PreferredNode* to zero.

Return Value

AllocateCommonBufferEx returns the virtual address of the memory allocated for the common buffer. If the buffer cannot be allocated, NULL is returned.

Remarks

AllocateCommonBufferEx is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter with the Version member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

AllocateCommonBufferEx is an extended version of the AllocateCommonBuffer routine. The following list summarizes the features that are available only in the extended version:

- The caller can specify a maximum logical address for the common buffer that is to be allocated.
- The caller can specify a preferred NUMA node in which the common buffer is to be allocated.

On computers with ARM or ARM 64-based processors, cache settings in the system ACPI have a higher precedence than the *CacheEnabled* parameter value passed by the driver. If the **ACPI _CCA** method indicates that the device is not cache coherent, the operating system disables caching even if the driver allocates cached memory with *CacheEnabled* set to TRUE.

On computers with ARM or ARM 64-based processors, the operating system allocates an uncached common buffer as Device Memory. For more information about the buffer, see sections A3.5.1, and A3.5.6 from the ARMv7 Architecture Reference Manual.

The processor does not permit misaligned access to Device Memory. Your driver must always access data from the common buffer by using naturally-aligned operations. Most kernel routines do not accept Device Memory as input parameters. For example, a network driver cannot pass Device Memory into NdisMIndicateReceiveNetBufferLists. If your driver needs to pass data from a DMA common buffer to a kernel routine, either allocate the buffer with *CacheEnabled* set to TRUE or copy the data from the uncached common buffer into a temporary pool allocation.

For more information about DMA operations that use a common buffer, see the following topics:

Using Common Buffers Using Common-Buffer Bus-Master DMA Using Common-Buffer System DMA

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	PASSIVE_LEVEL

See Also

AllocateCommonBuffer

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

MmGetPhysicalAddress

PBUILD_MDL_FROM_SCATTER_GATHER_LIST callback function

1/25/2019 • 2 minutes to read • Edit Online

The **BuildMdlFromScatterGatherList** routine builds an MDL from a scatter/gather list allocated by the system.

> **Note** This routine is reserved for system use.

Syntax

```
PBUILD_MDL_FROM_SCATTER_GATHER_LIST PbuildMdlFromScatterGatherList;

NTSTATUS PbuildMdlFromScatterGatherList(
  PDMA_ADAPTER DmaAdapter,
  PSCATTER_GATHER_LIST ScatterGather,
  PMDL OriginalMdl,
  PMDL *TargetMdl
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`ScatterGather`

Pointer to the SCATTER_GATHER_LIST structure passed to the driver's AdapterListControl routine.

`OriginalMdl`

Pointer to the original MDL that the driver used to build the scatter/gather list.

`*TargetMdl`

Pointer to a variable the routine uses to return the MDL created to hold the buffer described by the scatter/gather list. The value returned can be the same as *OriginalMdl*.

Return Value

BuildMdlFromScatterGatherList returns one of the following status codes:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded.

STATUS_INVALID_PARAMETER	The *OriginalMdl* parameter is **NULL**.
STATUS_INSUFFICIENT_RESOURCES	There is not enough memory available to allocate a new MDL.
STATUS_NONE_MAPPED	The system has already created a new MDL for the memory locations in the scatter/gather list. (This only happens if the routine is called twice on the same scatter/gather list.)

Remarks

BuildMdlFromScatterGatherList is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION2. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

When a driver creates a scatter/gather list to write to a device, the system can make a copy of the data to be written, and use that copy to perform the DMA operation. Use this routine to access the memory locations in the scatter/gather list, regardless of whether those locations are a copy.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.

Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

BuildScatterGatherList

DEVICE_DESCRIPTION

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

SCATTER_GATHER_LIST

PBUILD_SCATTER_GATHER_LIST callback function

1/25/2019 • 2 minutes to read • Edit Online

The **BuildScatterGatherList** routine prepares the system for a DMA operation, using a driver-supplied buffer to build the scatter/gather list.

Syntax

```
PBUILD_SCATTER_GATHER_LIST PbuildScatterGatherList;

NTSTATUS PbuildScatterGatherList(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PMDL Mdl,
  PVOID CurrentVa,
  ULONG Length,
  PDRIVER_LIST_CONTROL ExecutionRoutine,
  PVOID Context,
  BOOLEAN WriteToDevice,
  PVOID ScatterGatherBuffer,
  ULONG ScatterGatherLength
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`DeviceObject`

Pointer to the device object that represents the target device for the DMA operation.

`Mdl`

Pointer to the MDL that describes the buffer specified by the **MdlAddress** member of the current IRP.

`CurrentVa`

Pointer to the current virtual address in the MDL for the buffer to be mapped for a DMA transfer operation.

`Length`

Specifies the length, in bytes, of the buffer to be mapped.

`ExecutionRoutine`

Pointer to a driver-supplied AdapterListControl routine, which is called at IRQL = DISPATCH_LEVEL when the system DMA controller or bus-master adapter is available.

`Context`

Pointer to the driver-determined context passed to *ExecutionRoutine* when it is called.

`WriteToDevice`

Indicates the direction of the DMA transfer: **TRUE** for a transfer from the buffer to the device, and **FALSE** otherwise.

`ScatterGatherBuffer`

Pointer to the caller-supplied buffer that the routine fills with a SCATTER_GATHER_LIST structure.

`ScatterGatherLength`

Return Value

BuildScatterGatherList returns one of the following values:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The operation is completed successfully.
STA TUS _IN SUF FICI ENT _RE SO URC ES	The system has insufficient map registers available for the transfer.
STA TUS _BU FFE R_T OO_ SM ALL	The specified *Length* is too big to fit within the buffer.

Remarks

BuildScatterGatherList is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION2. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

BuildScatterGatherList performs the same operation as GetScatterGatherList, except that it uses the buffer supplied in the *ScatterGatherBuffer* parameter to hold the scatter/gather list that it creates. In contrast, **GetScatterGatherList** dynamically allocates a buffer to hold the scatter/gather list. If insufficient memory is available to allocate the buffer, **GetScatterGatherList** can fail with a STATUS_INSUFFICIENT_RESOURCES error. Drivers that must avoid this scenario can preallocate a buffer to hold the scatter/gather list, and use **BuildScatterGatherList** instead.

A driver can use the CalculateScatterGatherList routine to determine the size of buffer to allocate to hold the scatter/gather list.

The driver should retain the pointer to the scatter/gather list in *ScatterGatherBuffer* for use when the driver calls PutScatterGatherList. The driver must call **PutScatterGatherList** (which flushes the list) before it can access the data in the list.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch

See Also

BuildMdlFromScatterGatherList

CalculateScatterGatherList

DEVICE_DESCRIPTION

DEVICE_OBJECT

DMA_ADAPTER

DMA_OPERATIONS

GetScatterGatherList

IoGetDmaAdapter

PutScatterGatherList

SCATTER_GATHER_LIST

PBUILD_SCATTER_GATHER_LIST_EX callback function

1/25/2019 • 5 minutes to read • Edit Online

The **BuildScatterGatherListEx** routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.

> **Note** Do not call this routine for a system DMA device.

Syntax

```
PBUILD_SCATTER_GATHER_LIST_EX PbuildScatterGatherListEx;

NTSTATUS PbuildScatterGatherListEx(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PVOID DmaTransferContext,
  PMDL Mdl,
  ULONGLONG Offset,
  ULONG Length,
  ULONG Flags,
  PDRIVER_LIST_CONTROL ExecutionRoutine,
  PVOID Context,
  BOOLEAN WriteToDevice,
  PVOID ScatterGatherBuffer,
  ULONG ScatterGatherLength,
  PDMA_COMPLETION_ROUTINE DmaCompletionRoutine,
  PVOID CompletionContext,
  PVOID ScatterGatherList
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`DeviceObject`

A pointer to a DEVICE_OBJECT structure. This structure is the physical device object (PDO) that represents the target device for the requested DMA operation.

`DmaTransferContext`

A pointer to an initialized DMA transfer context. This context was initialized by a previous call to the InitializeDmaTransferContext routine. This context must be unique across all adapter allocation requests. To cancel a pending allocation request, the caller must supply the DMA transfer context for the request to the CancelAdapterChannel routine.

`Mdl`

A pointer to an MDL chain that describes the physical page layout for a collection of locked-down buffers in virtual memory. The scatter/gather list for the DMA transfer will use the region of this memory that is specified by the *Offset* and *Length* parameters. For more information about MDL chains, see Using MDLs.

`Offset`

The starting offset for the scatter/gather DMA transfer. This parameter is a byte offset from the start of the buffer in the first MDL in the MDL chain. If the MDLs in the MDL chain specify a total of N bytes of buffer space, valid values of *Offset* are in the range 0 to N–1.

`Length`

The size, in bytes, of the DMA transfer. If the MDL chain specifies a total of N bytes of buffer space, valid values of *Length* are in the range 1 to N–*Offset*.

`Flags`

The adapter channel allocation flags. The following flag is supported.

FLAG	MEANING
DMA_SYNCHRONOUS_CALLBACK	The **BuildScatterGatherListEx** routine is called synchronously. If this flag is set, and the required DMA resources are not immediately available, the call fails and returns STATUS_INSUFFICIENT_RESOURCES.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, the *ExecutionRoutine* parameter is optional and can be **NULL**. If this flag is not set, *ExecutionRoutine* must be a valid, non-**NULL** pointer. For more information about this flag, see the Remarks section.

`ExecutionRoutine`

A pointer to the driver-supplied AdapterListControl routine that initiates the DMA transfer for the driver. The I/O manager calls the *AdapterListControl* routine after the required resources are allocated for the adapter object. After the *AdapterListControl* routine returns, the I/O manager automatically frees the adapter object and the resources that were allocated for this object.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, *ExecutionRoutine* is optional and can be **NULL**. If *ExecutionRoutine* is **NULL**, the caller can use the resources allocated by **BuildScatterGatherListEx**. For more information, see the Remarks section.

`Context`

The driver-determined, adapter-control context. This context is passed to the AdapterListControl routine as the *Context* parameter.

`WriteToDevice`

The direction of the DMA transfer. Set this parameter to **TRUE** for a write operation, which transfers data from memory to the device. Set this parameter to **FALSE** for a read operation, which transfers data from the device to memory.

`ScatterGatherBuffer`

A pointer to a caller-allocated buffer into which the routine writes the scatter/gather list for the DMA transfer. This list begins with a SCATTER_GATHER_LIST structure, which is followed by a **SCATTER_GATHER_ELEMENT** array.

`ScatterGatherLength`

`DmaCompletionRoutine`

Not used. Set to **NULL**.

`CompletionContext`

Not used. Set to **NULL**.

`ScatterGatherList`

A pointer to a variable into which the routine writes a pointer to the scatter/gather list for the DMA transfer. This list begins with a SCATTER_GATHER_LIST structure, which contains a pointer to a **SCATTER_GATHER_ELEMENT** array. This output pointer always matches the *ScatterGatherBuffer* parameter value.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set and the *ExecutionRoutine* parameter is **NULL**, *ScatterGatherList* must be a valid, non-**NULL** pointer. If *ExecutionRoutine* is non-**NULL**, *ScatterGatherList* is optional and can be **NULL** if the calling driver does not require the scatter/gather list. The **BuildScatterGatherListEx** call fails if the **DMA_SYNCHRONOUS_CALLBACK** flag is set and *ScatterGatherList* and *ExecutionRoutine* are both **NULL**, or if the **DMA_SYNCHRONOUS_CALLBACK** flag is not set and *ExecutionRoutine* is **NULL**.

Return Value

BuildScatterGatherListEx returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETERS	The routine failed due to invalid parameter values passed by the caller.
STATUS_BUFFER_TOO_SMALL	The caller-supplied buffer in *ScatterGatherBuffer* is too small to contain the scatter/gather list.

STATUS_INSUFFICIENT_RESOURCES	The routine failed to allocate resources required for the DMA transfer.

Remarks

BuildScatterGatherListEx *is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a* DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

Use **BuildScatterGatherListEx** only for bus-master adapters. Do not use this routine for a system DMA adapter.

BuildScatterGatherListEx is similar to the GetScatterGatherListEx routine, except that it requires the caller to allocate the buffer for the scatter/gather list.

For example, a driver might preallocate one or more scatter/gather buffers during device initialization. Later, a **BuildScatterGatherListEx** call that uses such a buffer can succeed in conditions of low memory availability that might cause a **GetScatterGatherListEx** call to fail.

By default, **BuildScatterGatherListEx** returns asynchronously, without waiting for the requested resource allocation to complete. After this return, the caller can, if necessary, cancel the pending allocation request by calling the CancelAdapterChannel routine.

If the calling driver sets the **DMA_SYNCHRONOUS_CALLBACK** flag, the **BuildScatterGatherListEx** routine behaves as follows:

- If the requested resources are not immediately available, **BuildScatterGatherListEx** does not wait for resources, does not build a scatter/gather list, and does not call the *AdapterListControl* routine. Instead, **BuildScatterGatherListEx** fails and returns STATUS_INSUFFICIENT_RESOURCES.
- The driver is not required to supply an *AdapterListControl* routine if the **DMA_SYNCHRONOUS_CALLBACK** flag is set.
- If the driver supplies an *AdapterListControl* routine, the **DMA_SYNCHRONOUS_CALLBACK** flag indicates that this routine is to be called in the context of the calling thread, before **BuildScatterGatherListEx** returns.
- If the driver does not supply an *AdapterListControl* routine, the driver can use the allocated resources and scatter/gather list after **BuildScatterGatherListEx** returns. In this case, the driver must supply a valid, non-**NULL** *ScatterGatherList* pointer. In addition, after the driver-initiated DMA transfer completes, the driver must call the FreeAdapterObject routine to free the resources that **BuildScatterGatherListEx** allocated for the adapter object.

BuildScatterGatherListEx is an extended version of the BuildScatterGatherList routine. The following list summarizes the features that are available only in the extended version:

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL

See Also

AdapterListControl

AllocateAdapterChannelEx

CalculateScatterGatherList

DMA_OPERATIONS

DmaCompletionRoutine

FreeAdapterObject

GetScatterGatherList

GetScatterGatherListEx

MapTransferEx

PCALCULATE_SCATTER_GATHER_LIST_SIZE callback function

1/25/2019 • 2 minutes to read • Edit Online

The **CalculateScatterGatherList** routine calculates the size, in bytes, of scatter/gather list necessary to hold a given buffer.

Syntax

```
PCALCULATE_SCATTER_GATHER_LIST_SIZE PcalculateScatterGatherListSize;

NTSTATUS PcalculateScatterGatherListSize(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  PVOID CurrentVa,
  ULONG Length,
  PULONG ScatterGatherListSize,
  PULONG pNumberOfMapRegisters
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`Mdl`

Either **NULL** or a pointer to the MDL that contains the buffer.

`CurrentVa`

Pointer to the virtual address of the beginning of the buffer.

`Length`

Specifies the length of the buffer, in bytes.

`ScatterGatherListSize`

Pointer to the variable the routine uses to return the size of the scatter/gather list, in bytes.

`pNumberOfMapRegisters`

Return Value

CalculateScatterGatherList returns one of the following status codes.

RETURN CODE	DESCRIPTION

STA TUS _SU CCE SS	The values returned in *ScatterGatherListSize* and *NumberOfMapRegisters* are valid.
STA TUS _IN SUF FICI ENT _RE SO URC ES	The number of map registers required exceeds the number of map registers available.
STA TUS _BU FFE R_T OO_ SM ALL	The specified *Length* is too big to fit within the buffer.

Remarks

CalculateScatterGatherList is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION2. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

If the caller passes **NULL** for the *Mdl* parameter, the routine calculates the maximum possible size needed to hold a scatter/gather list for the specified buffer. If the caller specifies the MDL that contains the buffer in the *Mdl* parameter, the routine computes the actual size needed to hold the scatter/gather list.

A driver uses **CalculateScatterGatherList** to allocate a scatter/gather list buffer to pass to BuildScatterGatherList.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	Any level

See Also

BuildScatterGatherList

DEVICE_DESCRIPTION

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

SCATTER_GATHER_LIST

PCANCEL_ADAPTER_CHANNEL callback function

1/25/2019 • 2 minutes to read • Edit Online

The **CancelAdapterChannel** routine tries to cancel a pending request to allocate a DMA channel.

Syntax

```
PCANCEL_ADAPTER_CHANNEL PcancelAdapterChannel;

BOOLEAN PcancelAdapterChannel(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PVOID DmaTransferContext
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that is waiting for the pending resource allocation request to be granted. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine, and the caller passed this pointer to the AllocateAdapterChannelEx, GetScatterGatherListEx, or BuildScatterGatherListEx call that requested the resource allocation.

`DeviceObject`

A pointer to a DEVICE_OBJECT structure. This structure is the physical device object (PDO) that represents the target device for the DMA operation for which the DMA channel was requested.

`DmaTransferContext`

A pointer to a DMA transfer context. This parameter value must be the same DMA transfer context that was passed to the **AllocateAdapterChannelEx**, **GetScatterGatherListEx**, or **BuildScatterGatherListEx** call that requested the adapter channel allocation.

Return Value

CancelAdapterChannel returns **TRUE** if the pending allocation request is successfully canceled. The routine returns **FALSE** if the request could not be canceled (for example, because the DMA channel was allocated before the request could be canceled).

Remarks

CancelAdapterChannel
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

CancelAdapterChannel tries to cancel a pending allocation request that was made by an asynchronous call to an

allocation routine such as AllocateAdapterChannelEx, GetScatterGatherListEx, or BuildScatterGatherListEx. When called asynchronously, the allocation routine can return while the allocation request is still pending and before the driver-supplied execution routine (AdapterControl or AdapterListControl) is called. If **CancelAdapterChannel** successfully cancels the pending allocation request, the resources are not allocated and the execution routine is not called.

If a driver calls **AllocateAdapterChannelEx** to request resources for a driver-supplied *AdapterControl* routine, and then calls **CancelAdapterChannel** to cancel the request, **CancelAdapterChannel** can return one of the following values:

- **TRUE**, if the adapter object is waiting for the requested allocation when the **CancelAdapterChannel** call occurs. In this case, **CancelAdapterChannel** cancels the pending allocation request, and the driver's *AdapterControl* routine is not called.
- **FALSE**, if the *AdapterControl* routine was already called or is about to be called.

Similarly, if a driver calls **GetScatterGatherListEx** or **BuildScatterGatherListEx** to request resources for a driver-supplied *AdapterListControl* routine, and then calls **CancelAdapterChannel** to cancel the request, **CancelAdapterChannel** can return one of the following values:

- **TRUE**, if the adapter object is waiting for the requested allocation when the **CancelAdapterChannel** call occurs. In this case, **CancelAdapterChannel** cancels the pending allocation request, and the driver's *AdapterListControl* routine is not called.
- **FALSE**, if the *AdapterListControl* routine was already called or is about to be called.

CancelAdapterChannel can preemptively cancel future channel allocation requests that use the specified DMA transfer context. For example, if the caller supplies the same DMA transfer context to **CancelAdapterChannel** and **AllocateAdapterChannelEx**, and the **CancelAdapterChannel** call is successful and occurs before the **AllocateAdapterChannelEx** call, the allocation requested by the **AllocateAdapterChannelEx** call is automatically canceled.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AdapterControl

AdapterListControl

AllocateAdapterChannelEx

BuildScatterGatherListEx

CancelAdapterChannel

DMA_ADAPTER

DMA_OPERATIONS

GetScatterGatherListEx

IoGetDmaAdapter

PCANCEL_MAPPED_TRANSFER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **CancelMappedTransfer** routine cancels a system DMA transfer that is currently mapped to an adapter.

Syntax

```
PCANCEL_MAPPED_TRANSFER PcancelMappedTransfer;

NTSTATUS PcancelMappedTransfer(
  PDMA_ADAPTER DmaAdapter,
  PVOID DmaTransferContext
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that is waiting for the pending DMA transfer to complete. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine, and the caller passed this pointer to the MapTransferEx call that set up the map registers for the DMA transfer.

`DmaTransferContext`

A pointer to a DMA transfer context. This parameter value must be the same DMA transfer context that was passed to the AllocateAdapterChannelEx call that requested the adapter channel allocation.

Return Value

CancelMappedTransfer returns STATUS_SUCCESS if the call is successful. Otherwise, it returns an appropriate error code.

Remarks

CancelMappedTransfer
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a
DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

This routine marks the associated transfer context as canceled, which prevents any further transfers under this context. After a transfer is canceled, its completion routine is called, but the status for the transfer might be set to **DmaCancelled**.

Regardless of whether a system DMA transfer is canceled or completed, the FlushAdapterBuffers or FlushAdapterBuffersEx routine must always be called to flush any data that remains in the system DMA controller's internal cache at the end of a DMA transfer operation. After the transfer is canceled or completed, the **FlushAdapterBuffers*Xxx*** call must occur before either the DMA channel is released or the MapTransferEx routine is called to set up the map registers for a new DMA transfer.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AllocateAdapterChannelEx

DMA_ADAPTER

DMA_OPERATIONS

FlushAdapterBuffers

FlushAdapterBuffersEx

InitializeDmaTransferContext

IoGetDmaAdapter

MapTransferEx

PCI_MSIX_MASKUNMASK_ENTRY callback function

1/25/2019 • 2 minutes to read • Edit Online

The *MaskTableEntry* routine masks an interrupt in the MSI-X hardware interrupt table.

Syntax

```
PCI_MSIX_MASKUNMASK_ENTRY PciMsixMaskunmaskEntry;

NTSTATUS PciMsixMaskunmaskEntry(
  PVOID Context,
  ULONG TableEntry
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the PCI_MSIX_TABLE_CONFIG_INTERFACE structure for the interface.

`TableEntry`

The index of the table entry in the MSI-X hardware interrupt table.

Return Value

The *MaskTableEntry* routine might return one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation completed successfully.
STATUS_INVALID_PARAMETER	The *TableEntry* parameter is invalid.

Remarks

If a table entry is masked, the device does not generate any interrupts that correspond to that table entry.

You can unmask the table entry by calling UnmaskTableEntry.

Requirements

Minimum supported client	Available in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of the Windows operating system.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DIRQL"

See Also

PCI_MSIX_TABLE_CONFIG_INTERFACE

UnmaskTableEntry

PCI_MSIX_SET_ENTRY callback function

1/25/2019 • 2 minutes to read • Edit Online

The *SetTableEntry* routine sets the message ID for a table entry in the MSI-X hardware interrupt table.

Syntax

```
PCI_MSIX_SET_ENTRY PciMsixSetEntry;

NTSTATUS PciMsixSetEntry(
  PVOID Context,
  ULONG TableEntry,
  ULONG MessageNumber
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the PCI_MSIX_TABLE_CONFIG_INTERFACE structure for the interface.

`TableEntry`

The index of the table entry in the MSI-X hardware interrupt table.

`MessageNumber`

The message ID for the interrupt. This value is also the index for the interrupt's entry in the **MessageInfo** member of the IO_INTERRUPT_MESSAGE_INFO structure that describes the driver's message-signaled interrupts. The IoConnectInterruptEx function supplies a pointer to this structure.

Return Value

The *SetTableEntry* routine might return one of the following NTSTATUS values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation completed successfully.

STATUS_INVALID_PARAMETER	The device does not use MSI-X, or the *TableEntry* or *MessageNumber* parameters do not correspond to interrupt resources that are assigned to the device.

Remarks

By default, the operating system assigns the index of the table entry as the message ID for the interrupt. If there are more table entries than messages, the system sets the remaining table entries to correspond to message zero. The driver can use the *SetTableEntry* routine to assign a different message ID.

Requirements

Minimum supported client	Available in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of the Windows operating system.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DIRQL"

See Also

IO_INTERRUPT_MESSAGE_INFO

IoConnectInterruptEx

PCI_MSIX_TABLE_CONFIG_INTERFACE

PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ClfsAdvanceTailCallback* function advances the base log sequence number (LSN) of the client's log.

Syntax

```
PCLFS_CLIENT_ADVANCE_TAIL_CALLBACK PclfsClientAdvanceTailCallback;

NTSTATUS PclfsClientAdvanceTailCallback(
  PLOG_FILE_OBJECT LogFile,
  PCLFS_LSN TargetLsn,
  PVOID ClientData
)
{...}
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log stream whose tail should be advanced.

`TargetLsn`

A pointer to a CLFS_LSN structure that contains the LSN that the client should advance its tail to or beyond.

`ClientData`

A pointer to client-supplied information. You specify this data in the **AdvanceTailCallbackData** member of the CLFS_MGMT_CLIENT_REGISTRATION structure.

Return Value

The *ClfsAdvanceTailCallback* function should return either STATUS_PENDING or an error status. A return value of STATUS_PENDING indicates that the request to move the client's log tail will be completed asynchronously. The **ClfsMgmtAdvanceTailCallback** function must not return STATUS_SUCCESS, even if it completes synchronously.

Remarks

The *ClfsAdvanceTailCallback* function is called when CLFS management requests that the client advance its log tail.

The *ClfsAdvanceTailCallback* function must only perform a minimal amount of processing before returning. For example, the *ClfsAdvanceTailCallback* function might follow this procedure:

1. If the request cannot be processed, return an error status.
2. If the request can be processed, create and queue a work item to perform the actions that are required to move the client's tail, and then return STATUS_PENDING. For more information on queuing a worker thread, see System Worker Threads.

When a client calls the ClfsMgmtRegisterManagedClient routine to register with CLFS management, the client provides both a pointer to the *ClfsAdvanceTailCallback* function and the custom data that will be passed as a parameter to the *ClfsAdvanceTailCallback* function when this function is called.

Requirements

Minimum supported client	Supported in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= APC_LEVEL

See Also

CLFS_MGMT_CLIENT_REGISTRATION

ClfsAdvanceLogBase

ClfsMgmtRegisterManagedClient

ClfsMgmtTailAdvanceFailure

PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ClfsLogGrowthCompleteCallback* function implements the actions that the client will take when space is freed in a log that had previously been full.

Syntax

```
PCLFS_CLIENT_LFF_HANDLER_COMPLETE_CALLBACK PclfsClientLffHandlerCompleteCallback;

void PclfsClientLffHandlerCompleteCallback(
  PLOG_FILE_OBJECT LogFile,
  NTSTATUS OperationStatus,
  BOOLEAN LogIsPinned,
  PVOID ClientData
)
{...}
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log stream whose log has finished growing.

`OperationStatus`

The status of the client's earlier request to handle a log file full condition. This will be either STATUS_SUCCESS or an error status that indicates the reason that the request could not be completed successfully.

`LogIsPinned`

If **TRUE**, specifies that the log tail is currently pinned; **FALSE** if the log tail is not pinned.

`ClientData`

A pointer to client-supplied information.

Return Value

None

Remarks

Because a call to ClfsMgmtHandleLogFileFull completes asynchronously, any actions that the client will take when the log file full condition has been handled belong in the *ClfsLogGrowthCompleteCallback* function.

If a call to the **ClfsMgmtHandleLogFileFull** routine returned returns STATUS_PENDING, then the *ClfsLogGrowthCompleteCallback* function will be invoked when the operation completes. If a different status was returned, then the *ClfsLogGrowthCompleteCallback* function will not be invoked.

The *ClfsLogGrowthCompleteCallback* function should only perform a minimal amount of processing before returning. For example, the *ClfsLogGrowthCompleteCallback* function might follow the following procedure:

1. If the value of the *OperationStatus* parameter is an error status, return.
2. If the value of the *OperationStatus* parameter is STATUS_SUCCESS, create and queue a work item to inform the client that it can resume processing.

When a client uses the ClfsMgmtRegisterManagedClient routine to register with CLFS management, the client provides both a pointer to the *ClfsLogGrowthCompleteCallback* function and the custom data that will be passed as a parameter to the *ClfsLogGrowthCompleteCallback* function when this function is called.

Requirements

Minimum supported client	Supported in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= APC_LEVEL

See Also

ClfsMgmtHandleLogFileFull

ClfsMgmtRegisterManagedClient

PCLFS_CLIENT_LOG_UNPINNED_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ClfsLogUnpinnedCallback* function implements any actions that the client will take when a log that had previously been pinned becomes unpinned.

Syntax

```
PCLFS_CLIENT_LOG_UNPINNED_CALLBACK PclfsClientLogUnpinnedCallback;

void PclfsClientLogUnpinnedCallback(
  PLOG_FILE_OBJECT LogFile,
  PVOID ClientData
)
{...}
```

Parameters

`LogFile`

A pointer to a LOG_FILE_OBJECT structure that represents the CLFS log stream whose tail has become unpinned.

`ClientData`

A pointer to client-supplied information. You specify this data in the **LogUnpinnedCallbackData** member of the CLFS_MGMT_CLIENT_REGISTRATION structure.

Return Value

None

Remarks

If a client called the ClfsMgmtHandleLogFileFull routine and CLFS management subsequently called the client's ClfsLogGrowthCompleteCallback function with a value of **TRUE** for the *LogIsPinned* parameter, then the *ClfsLogUnpinnedCallback* function will be invoked when the log becomes unpinned.

When a client uses the ClfsMgmtRegisterManagedClient routine to register with CLFS management, the client provides both a pointer to the *ClfsLogUnpinnedCallback* function and the custom data that will be passed as a parameter to the *ClfsLogUnpinnedCallback* function when this function is called.

The *ClfsLogUnpinnedCallback* function should only perform a minimal amount of processing before returning. For example, it might create and queue a work item to inform the client that it can resume processing.

The *ClfsLogGrowthCompleteCallback* function is called before the *ClfsLogUnpinnedCallback* function.

Requirements

Minimum supported client	Supported in Windows Server 2003 R2, Windows Vista, and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= APC_LEVEL

See Also

CLFS_MGMT_CLIENT_REGISTRATION

ClfsMgmtHandleLogFileFull

PCONFIGURE_ADAPTER_CHANNEL callback function

1/25/2019 • 2 minutes to read • Edit Online

The **ConfigureAdapterChannel** routine invokes a custom function that is implemented by the DMA controller represented by an adapter object.

Syntax

```
PCONFIGURE_ADAPTER_CHANNEL PconfigureAdapterChannel;

NTSTATUS PconfigureAdapterChannel(
  PDMA_ADAPTER DmaAdapter,
  ULONG FunctionNumber,
  PVOID Context
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`FunctionNumber`

The number of the custom function to select. For more information, see the Remarks section.

`Context`

A pointer to the configuration parameters for the custom function that is specified by *FunctionNumber*. The function is implemented by the DMA controller and can be accessed by a device driver through the adapter object. The DMA controller and the device driver must agree on the meaning of these configuration parameters. The data that *Context* points to is opaque to the operating system.

Return Value

ConfigureAdapterChannel returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status code.

RETURN VALUE	DESCRIPTION
STA TUS _NO T_I MPL EME NTE D	The DMA extension does not implement the function specified by *FunctionNumber*.

Remarks

ConfigureAdapterChannel
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a
DMA_OPERATIONS*structure*.Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

Use **ConfigureAdapterChannel** only for system DMA adapters. Do not use this routine for a bus-master adapter.

A particular system DMA controller might have special hardware features that device drivers can access through a set of one or more custom functions. A controller might implement more than one custom function, in which case each function is identified by a different function number. These functions abstract the DMA hardware features so that the same functions can be supported on different hardware platforms.

Typically, a custom function sets state information in the DMA controller to configure hardware features for use in subsequent DMA transfers.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

PDEVICE_RESET_HANDLER callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DeviceReset* routine is used to reset and recover a malfunctioning device.

Syntax

```
PDEVICE_RESET_HANDLER PdeviceResetHandler;

NTSTATUS PdeviceResetHandler(
  PVOID InterfaceContext,
  DEVICE_RESET_TYPE ResetType,
  ULONG Flags,
  PVOID ResetParameters
)
{...}
```

Parameters

`InterfaceContext`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the DEVICE_RESET_INTERFACE_STANDARD structure for the interface.

`ResetType`

The type of reset being requested. Set this parameter to one of the following DEVICE_RESET_TYPE enumeration values.

- **FunctionLevelDeviceReset**. Specify this value to request a function-level reset, which is restricted to a specific device.
- **PlatformLevelDeviceReset**. Specify this value to request a platform-level reset, which affects a specific device and all other devices that are connected to it via the same power rail or reset line.

For more information about how function-level and platform-level resets are implemented in the device stack, see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD.

`Flags`

Set to 0. Currently, no flags are defined for this routine.

`ResetParameters`

If the caller is requesting a function-level device reset, this optional parameter can point to a FUNCTION_LEVEL_DEVICE_RESET_PARAMETERS structure that specifies a callback routine that is called when the reset is completed.

Return Value

This routine returns STATUS_SUCCESS if the requested operation succeeds. Otherwise, it returns an appropriate an NTSTATUS error code.

Remarks

If a function driver detects that the device is not functioning correctly, it should first attempt a function-level reset. If a function-level reset does not fix the issue, then the driver can attempt a more invasive platform-level reset, but a platform-level reset should only be used as the final option.

For more information about function-level and platform-level resets, see Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Desktop
Header	wdm.h

See Also

Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD

IRP_MN_QUERY_INTERFACE

PFLUSH_ADAPTER_BUFFERS callback function

1/25/2019 • 2 minutes to read • Edit Online

The **FlushAdapterBuffers** routine flushes any data remaining in the system DMA controller's internal cache or in a bus-master adapter's internal cache at the end of a DMA transfer operation.

Syntax

```
PFLUSH_ADAPTER_BUFFERS PflushAdapterBuffers;

BOOLEAN PflushAdapterBuffers(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  PVOID MapRegisterBase,
  PVOID CurrentVa,
  ULONG Length,
  BOOLEAN WriteToDevice
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`Mdl`

Pointer to the MDL that describes the buffer previously passed in the driver's call to **MapTransfer**.

`MapRegisterBase`

Specifies the map registers allocated for the DMA operation. The system passes this value to the driver's AdapterControl routine.

`CurrentVa`

Pointer to the current virtual address in the buffer, described by the *Mdl*, where the I/O operation occurred. This value must be the same as the initial *CurrentVa* value passed to MapTransfer.

`Length`

Specifies the length, in bytes, of the buffer.

`WriteToDevice`

Specifies the direction of the DMA transfer operation: **TRUE** for a transfer from a buffer in system memory to the driver's device.

Return Value

FlushAdapterBuffers returns **TRUE** if any data remaining in the DMA controller's or bus-master adapter's internal cache has been successfully flushed into system memory or out to the device.

Remarks

FlushAdapterBuffers is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

To ensure that a DMA transfer is complete, every driver that performs DMA operations must call **FlushAdapterBuffers** before completing the IRP that requested the DMA transfer and before freeing the map registers.

A driver can get the initial *CurrentVa* for the start of a packet-based DMA transfer by calling MmGetMdlVirtualAddress. However, the value returned is an index into the *Mdl*, rather than a valid virtual address. If the driver must split a large transfer request into more than one DMA operation, it must update *CurrentVa* and *Length* for each DMA operation.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

AllocateAdapterChannel

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

KeFlushIoBuffers

MapTransfer

MmGetMdlVirtualAddress

PFLUSH_ADAPTER_BUFFERS_EX callback function

1/25/2019 • 3 minutes to read • Edit Online

The **FlushAdapterBuffersEx** routine flushes any data that remains in the data cache at the end of a DMA transfer operation performed by a system DMA controller or bus-master device.

Syntax

```
PFLUSH_ADAPTER_BUFFERS_EX PflushAdapterBuffersEx;

NTSTATUS PflushAdapterBuffersEx(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  PVOID MapRegisterBase,
  ULONGLONG Offset,
  ULONG Length,
  BOOLEAN WriteToDevice
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's system DMA channel or bus-master device. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`Mdl`

A pointer to the MDL chain that describes the pages of memory that are to be flushed. Set this parameter to point to the MDL chain that was used for the DMA transfer. For more information, see the description of the *Mdl* parameter in GetScatterGatherListEx, BuildScatterGatherListEx, or MapTransferEx.

`MapRegisterBase`

A handle to the map registers that were previously allocated for the adapter object.

`Offset`

The starting offset relative to the start of the MDL chain to use for the flush operation. Set this parameter to the value that was used as the starting offset for the DMA transfer. For more information, see the description of the *Offset* parameter in GetScatterGatherListEx, BuildScatterGatherListEx, or MapTransferEx.

`Length`

The number of bytes of data to flush. Set this parameter to the value that was used as the transfer length for the DMA transfer. For more information, see the description of the *Length* parameter in GetScatterGatherListEx, BuildScatterGatherListEx, or MapTransferEx.

`WriteToDevice`

The direction of the DMA transfer. Set this parameter to the value that was used as the transfer direction for the DMA transfer. For more information, see the description of the *WriteToDevice* parameter in

GetScatterGatherListEx, BuildScatterGatherListEx, or MapTransferEx.

Return Value

FlushAdapterBuffersEx returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ PAR AM ETE RS	The routine failed due to invalid parameter values passed by the caller.

Remarks

FlushAdapterBuffersEx
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS*structure*.Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

FlushAdapterBuffersEx enables processor cache coherency in hardware platforms that do not implement hardware-enforced cache coherency (bus snooping). In addition, for a system DMA transfer, **FlushAdapterBuffersEx** flushes any data that remains in the system DMA controller's internal cache.

The driver that initiates a scatter/gather DMA transfer must ensure that all of the transferred data is flushed from the cache after the transfer completes. The driver should call **FlushAdapterBuffersEx** before the driver completes the IRP that requested the DMA transfer, and before the driver frees the map registers. For more information, see Flushing Cached Data during DMA Operations.

FlushAdapterBuffersEx is an extended version of the FlushAdapterBuffers routine. **FlushAdapterBuffersEx** can flush an entire scatter/gather list in one call. In contrast, to use **FlushAdapterBuffers** to flush a scatter/gather list requires a separate call for each MDL in the MDL chain.

FlushAdapterBuffersEx can be used in place of **FlushAdapterBuffers** to flush the buffer for a one-packet DMA transfer. In this case, the MDL chain that the *Mdl* parameter points to contains only one MDL. A benefit of using **FlushAdapterBuffersEx** in this way is that the caller can specify an offset into the MDL.

If **FlushAdapterBuffersEx** is called before the DMA transfer operation finishes, this call might cancel the transfer before it can finish or might cause undefined behavior.

Requirements

Minimum supported client	Available starting with Windows 8.

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

BuildScatterGatherListEx

DMA_ADAPTER

DMA_OPERATIONS

FlushAdapterBuffers

GetScatterGatherListEx

IoGetDmaAdapter

MapTransferEx

PFREE_ADAPTER_CHANNEL callback function

1/25/2019 • 2 minutes to read • Edit Online

The **FreeAdapterChannel** routine releases the system DMA controller when a driver has completed all DMA operations necessary to satisfy the current IRP.

Syntax

```
PFREE_ADAPTER_CHANNEL PfreeAdapterChannel;

void PfreeAdapterChannel(
  PDMA_ADAPTER DmaAdapter
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

Return Value

None

Remarks

FreeAdapterChannel is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

After a driver has transferred all the data and called FlushAdapterBuffers, it calls **FreeAdapterChannel** to release the system DMA controller that was previously allocated with a call to **AllocateAdapterChannel**.

FreeAdapterChannel frees any map registers that were allocated by an earlier call to **AllocateAdapterChannel**. A driver calls this routine only if its AdapterControl routine returns **KeepObject**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL

DDI compliance rules	IrqlDispatch, IrqlDispatch(storport)

See Also

AllocateAdapterChannel

DMA_ADAPTER

DMA_OPERATIONS

FlushAdapterBuffers

FreeMapRegisters

IoGetDmaAdapter

MapTransfer

PFREE_ADAPTER_OBJECT callback function

1/25/2019 • 2 minutes to read • Edit Online

The **FreeAdapterObject** routine releases the specified adapter object after a driver has completed all DMA operations.

Syntax

```
PFREE_ADAPTER_OBJECT PfreeAdapterObject;

void PfreeAdapterObject(
  PDMA_ADAPTER DmaAdapter,
  IO_ALLOCATION_ACTION AllocationAction
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that is to be released. This object represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`AllocationAction`

The type of deallocation that the calling driver requests. Set this parameter to an IO_ALLOCATION_ACTION enumeration value. For more information, see the Remarks section.

Return Value

None

Remarks

FreeAdapterObject is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

If *AllocationAction* = **DeallocateObject**, the **FreeAdapterObject** call releases the adapter object, and releases any map registers that might be allocated for this object.

If *AllocationAction* = **DeallocateObjectKeepRegisters**, the **FreeAdapterObject** call releases the adapter object, but does not release any map registers. In this case, the driver must explicitly release any allocated map registers by calling the FreeMapRegisters routine.

If *AllocationAction* = **KeepObject**, the **FreeAdapterObject** call has no effect.

FreeAdapterObject can be used to deallocate DMA resources that are allocated by calls to the AllocateAdapterChannelEx, GetScatterGatherListEx, and BuildScatterGatherListEx routines. If the driver calls one

of these routines synchronously, the *ExecutionRoutine* parameter is optional and can be omitted. In this case, the driver can use the adapter object and map registers that are allocated by the call. After the driver initiates the DMA transfer, the driver can call **FreeAdapterObject** to deallocate the adapter object and, if necessary, the map registers.

The FreeAdapterChannel routine is similar to **FreeAdapterObject**. **FreeAdapterChannel** always releases the adapter object and any allocated map registers. This behavior is similar to that of **FreeAdapterObject** if *AllocationAction* = **DeallocateObject**.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AdapterControl

AdapterListControl

AllocateAdapterChannel

AllocateAdapterChannelEx

BuildScatterGatherListEx

DMA_ADAPTER

DMA_OPERATIONS

FreeAdapterChannel

FreeMapRegisters

GetScatterGatherListEx

IO_ALLOCATION_ACTION

IoGetDmaAdapter

PFREE_COMMON_BUFFER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **FreeCommonBuffer** routine frees a common buffer allocated by AllocateCommonBuffer, along with all resources the buffer uses.

Syntax

```
PFREE_COMMON_BUFFER PfreeCommonBuffer;

void PfreeCommonBuffer(
  PDMA_ADAPTER DmaAdapter,
  ULONG Length,
  PHYSICAL_ADDRESS LogicalAddress,
  PVOID VirtualAddress,
  BOOLEAN CacheEnabled
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`Length`

Specifies the number of bytes to deallocate.

`LogicalAddress`

Specifies the logical address of the allocated memory range.

`VirtualAddress`

Pointer to the corresponding virtual address of the allocated memory range.

`CacheEnabled`

Indicates whether the allocated memory is cached.

Return Value

None

Remarks

FreeCommonBuffer is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

To release a common buffer, a driver calls **FreeCommonBuffer** to unmap both its logical and virtual addresses. The parameters passed to **FreeCommonBuffer** must match exactly those passed to and returned from

AllocateCommonBuffer. A driver cannot free part of an allocated common buffer.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlDispatch

See Also

AllocateCommonBuffer

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

PFREE_MAP_REGISTERS callback function

1/25/2019 • 2 minutes to read • Edit Online

The **FreeMapRegisters** routine releases a set of map registers that were saved from a call to AllocateAdapterChannel.

Syntax

```
PFREE_MAP_REGISTERS PfreeMapRegisters;

void PfreeMapRegisters(
  PDMA_ADAPTER DmaAdapter,
  PVOID MapRegisterBase,
  ULONG NumberOfMapRegisters
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`MapRegisterBase`

Specifies the map registers allocated for the DMA operation. The system passes this value to the driver's AdapterControl routine.

`NumberOfMapRegisters`

Specifies the number of map registers to be released. This value must match the number specified in an earlier call to AllocateAdapterChannel.

Return Value

None

Remarks

FreeMapRegisters is not a system routine that can be called directly by name. This routine is only callable by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

When the driver of a bus-master device has completed the current packet-based DMA transfer request, it calls **FreeMapRegisters** to release the map registers previously allocated by a call to AllocateAdapterChannel and retained because its *AdapterControl* routine returned **DeallocateObjectKeepRegisters**. The driver must call **FreeMapRegisters** after calling FlushAdapterBuffers.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, IrqlDispatch(storport)

See Also

AllocateAdapterChannel

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

MapTransfer

PGET_DMA_ADAPTER_INFO callback function

1/25/2019 • 2 minutes to read • Edit Online

The **GetDmaAdapterInfo** routine retrieves information about the hardware capabilities of a system DMA channel.

Syntax

```
PGET_DMA_ADAPTER_INFO PgetDmaAdapterInfo;

NTSTATUS PgetDmaAdapterInfo(
  PDMA_ADAPTER DmaAdapter,
  PDMA_ADAPTER_INFO AdapterInfo
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`AdapterInfo`

A pointer to a caller-allocated DMA_ADAPTER_INFO structure. The routine writes information about the system DMA controller into this structure. The caller must set the **Version** member of this structure to DMA_ADAPTER_INFO_VERSION1 before calling **GetDmaAdapterInfo**.

Return Value

GetDmaAdapterInfo returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STATUS_NOT_SUPPORTED	The routine does not support the specified version of the **DMA_ADAPTER_INFO*_XXX*** structure.

Remarks

GetDmaAdapterInfo
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a
DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the

Version member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

GetDmaAdapterInfo retrieves the following information:

- The maximum number of elements in a scatter/gather list that the DMA controller can process in a single scatter/gather DMA operation.
- The width, in bits, of a DMA address.

For information about the DMA adapter information that is provided by version 1 of the **DMA_ADAPTER_INFO_*XXX*** structure, see DMA_ADAPTER_INFO_V1.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

DMA_ADAPTER

DMA_ADAPTER_INFO

DMA_OPERATIONS

IoGetDmaAdapter

PGET_DMA_ALIGNMENT callback function

1/25/2019 • 2 minutes to read • Edit Online

The **GetDmaAlignment** routine returns the alignment requirements of the DMA system.

Syntax

```
PGET_DMA_ALIGNMENT PgetDmaAlignment;

ULONG PgetDmaAlignment(
  PDMA_ADAPTER DmaAdapter
)
{...}
```

Parameters

`DmaAdapter`

A pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

Return Value

GetDmaAlignment returns the alignment requirements, in bytes, of the DMA system. The starting address and length of DMA buffers must be a multiple of this value.

Remarks

GetDmaAlignment is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

A driver can call this routine to determine alignment requirements for DMA buffers it allocates. The returned value should be used to set the **AlignmentRequirement** field in the device object. A driver may need to increase this value because of additional hardware device restrictions. For more information, see Initializing a Device Object.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlDispatch

See Also

DEVICE_OBJECT

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

PGET_DMA_TRANSFER_INFO callback function

1/25/2019 • 3 minutes to read • Edit Online

The **GetDmaTransferInfo** routine calculates the allocation requirements for a scatter/gather DMA transfer.

Syntax

```
PGET_DMA_TRANSFER_INFO PgetDmaTransferInfo;

NTSTATUS PgetDmaTransferInfo(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  ULONGLONG Offset,
  ULONG Length,
  BOOLEAN WriteOnly,
  PDMA_TRANSFER_INFO TransferInfo
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`Mdl`

A pointer to the MDL chain that describes the pages of memory that are to be transferred. For more information, see the Remarks section.

`Offset`

The starting offset for the scatter/gather DMA transfer. This parameter is a byte offset from the start of the buffer in the first MDL in the MDL chain. If the MDLs in the MDL chain specify a total of N bytes of buffer space, valid values of *Offset* are in the range 0 to N–1.

`Length`

The length, in bytes, of the DMA transfer. If the MDL chain specifies a total of N bytes of buffer space, valid values of *Length* are in the range 1 to N–*Offset*.

`WriteOnly`

Indicates whether the transfer is a write-only operation (that is, a transfer to the device from memory). If *WriteOnly* is TRUE, information returned about the transfer might not be valid for a read transaction. This parameter might indicate that additional cache control operations are required at buffer boundaries to enforce cache coherency.

`TransferInfo`

A pointer to a caller-allocated DMA_TRANSFER_INFO structure. The caller must set the **Version** member of the structure to DMA_TRANSFER_INFO_VERSION1 before calling **GetDmaTransferInfo**.

Return Value

GetDmaTransferInfo returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STA TUS _NO T_S UPP ORT ED	The routine does not support the specified version of the **DMA_TRANSFER_INFO_*XXX*** structure.

Remarks

GetDmaTransferInfo
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS*structure*.Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

Use **GetDmaTransferInfo** to calculate the size of the scatter/gather buffer to allocate for the BuildScatterGatherListEx and MapTransferEx routines.

GetDmaTransferInfo replaces the CalculateScatterGatherList routine and is more convenient to use with **BuildScatterGatherListEx**.

The *Mdl*, *Offset*, and *Length* parameters together specify a collection of physical memory regions to use as a buffer for a scatter/gather DMA transfer. Each physical memory region is typically a page or part of a page. **GetDmaTransferInfo** determines the allocation requirements for this transfer. These requirements include the following:

- The number of elements in the scatter/gather list. Each element is a **SCATTER_GATHER_ELEMENT** structure that describes a physically contiguous block of memory.
- The amount of memory to allocate to hold the specified scatter/gather list. (This memory allocation includes the space required for the scatter/gather list that describes the I/O buffer but not for the I/O buffer itself.)
- The number of map registers required to translate the physical addresses in the scatter/gather list to logical addresses.

For information about the DMA transfer information that is provided by version 1 of the **DMA_TRANSFER_INFO_*XXX*** structure, see DMA_TRANSFER_INFO_V1.

An MDL describes the physical memory pages that underlie a locked-down, contiguous block of virtual memory. Typically, these physical memory pages are non-contiguous. An MDL chain is an ordered collection of MDLs that describes memory that can be used to buffer I/O data. Typically, the virtual memory regions described by the MDLs in the chain are non-contiguous. For more information about MDLs and MDL chains, see Using MDLs.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

BuildScatterGatherListEx

CalculateScatterGatherList

DMA_OPERATIONS

DMA_TRANSFER_INFO

DMA_TRANSFER_INFO_V1

IoGetDmaAdapter

MapTransferEx

PGET_SCATTER_GATHER_LIST callback function

1/25/2019 • 2 minutes to read • Edit Online

The **GetScatterGatherList** routine prepares the system for a DMA scatter/gather operation on behalf of the target device object, through either the system DMA controller or a bus-master adapter.

Syntax

```
PGET_SCATTER_GATHER_LIST PgetScatterGatherList;

NTSTATUS PgetScatterGatherList(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PMDL Mdl,
  PVOID CurrentVa,
  ULONG Length,
  PDRIVER_LIST_CONTROL ExecutionRoutine,
  PVOID Context,
  BOOLEAN WriteToDevice
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`DeviceObject`

Pointer to the device object that represents the target device for the DMA operation.

`Mdl`

Pointer to the MDL that describes the buffer at *MdlAddress* in the current IRP.

`CurrentVa`

Pointer to the current virtual address in the MDL for the buffer to be mapped for a DMA transfer operation.

`Length`

Specifies the length, in bytes, to be mapped.

`ExecutionRoutine`

Pointer to a driver-supplied AdapterListControl routine, which is called at DISPATCH_LEVEL when the system DMA controller or bus-master adapter is available.

`Context`

Pointer to the driver-determined context passed to the driver's *AdapterListControl* routine when it is called.

`WriteToDevice`

Indicates the direction of the DMA transfer: **TRUE** for a transfer from the buffer to the device, and **FALSE**

otherwise.

Return Value

This routine can return one of the following NTSTATUS values.

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation succeeded.
STATUS_INSUFFICIENT_RESOURCES	The routine could not allocate sufficient memory or the number of map registers required for the transfer is larger than the value returned by IoGetDmaAdapter.
STATUS_BUFFER_TOO_SMALL	The buffer is too small for the requested transfer.

Remarks

The **GetScatterGatherList** routine dynamically allocates a buffer to hold the scatter/gather list. For possible NTSTATUS values if the buffer allocation fails, see the return value.

GetScatterGatherList is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

As soon as the appropriate DMA channel and any necessary map registers are available, **GetScatterGatherList** creates a scatter/gather list, initializes the map registers, and then calls the driver-supplied AdapterListControl routine to carry out the I/O operation.

GetScatterGatherList combines the actions of the AllocateAdapterChannel and MapTransfer routines for drivers that perform scatter/gather DMA. **GetScatterGatherList** determines how many map registers are required for the transfer, allocates the map registers, maps the buffers for DMA, and fills in the scatter/gather list. It then calls the supplied AdapterListControl routine, passing a pointer to the scatter/gather list in *ScatterGather*. The driver should retain this pointer for use when calling PutScatterGatherList. Note that **GetScatterGatherList** does not have the queuing restrictions that apply to **AllocateAdapterChannel**.

In its *AdapterListControl* routine, the driver should perform the I/O. On return from the driver-supplied routine,

GetScatterGatherList keeps the map registers but frees the DMA adapter structure. The driver must call **PutScatterGatherList** (which flushes the buffers) before it can access the data in the buffer.

This routine can handle chained MDLs, provided that the total number of map registers required by all chained MDLs does not exceed the number of map registers that are available.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Not supported in Windows 98 or Windows Me.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, IrqlDispatch(storport)

See Also

AllocateAdapterChannel

DEVICE_OBJECT

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

PutScatterGatherList

SCATTER_GATHER_LIST

PGET_SCATTER_GATHER_LIST_EX callback function

1/25/2019 • 6 minutes to read • Edit Online

The **GetScatterGatherListEx** routine allocates the resources that are required for a DMA transfer, builds a scatter/gather list, and calls the driver-supplied AdapterListControl routine to initiate the DMA transfer.

> **Note** Do not call this routine for a system DMA device.

Syntax

```
PGET_SCATTER_GATHER_LIST_EX PgetScatterGatherListEx;

NTSTATUS PgetScatterGatherListEx(
  PDMA_ADAPTER DmaAdapter,
  PDEVICE_OBJECT DeviceObject,
  PVOID DmaTransferContext,
  PMDL Mdl,
  ULONGLONG Offset,
  ULONG Length,
  ULONG Flags,
  PDRIVER_LIST_CONTROL ExecutionRoutine,
  PVOID Context,
  BOOLEAN WriteToDevice,
  PDMA_COMPLETION_ROUTINE DmaCompletionRoutine,
  PVOID CompletionContext,
  PSCATTER_GATHER_LIST *ScatterGatherList
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`DeviceObject`

A pointer to a DEVICE_OBJECT structure. This structure is the physical device object (PDO) that represents the target device for the requested DMA operation.

`DmaTransferContext`

A pointer to an initialized DMA transfer context. This context was initialized by a previous call to the InitializeDmaTransferContext routine. This context must be unique across all adapter allocation requests. To cancel a pending allocation request, the caller must supply the DMA transfer context for the request to the CancelAdapterChannel routine.

`Mdl`

A pointer to an MDL chain that describes the physical page layout for a collection of locked-down buffers in virtual memory. The scatter/gather list for the DMA transfer will use the region of this memory that is specified by the *Offset* and *Length* parameters. For more information about MDL chains, see Using MDLs.

`Offset`

The starting offset for the scatter/gather DMA transfer. This parameter is a byte offset from the start of the buffer in the first MDL in the MDL chain. If the MDLs in the MDL chain specify a total of N bytes of buffer space, valid values of *Offset* are in the range 0 to N−1.

`Length`

The length, in bytes, of the DMA transfer. If the MDL chain specifies a total of N bytes of buffer space, valid values of *Length* are in the range 1 to N−*Offset*.

`Flags`

The adapter channel allocation flags. The following flag is supported.

FLAG	MEANING
DM A_S YN CH RO NO US_ CAL LBA CK	The **GetScatterGatherListEx** routine is called synchronously. If this flag is set, and the required DMA resources are not immediately available, the call fails and returns STATUS_INSUFFICIENT_RESOURCES.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, the *ExecutionRoutine* parameter is optional and can be NULL. If this flag is not set, *ExecutionRoutine* must be a valid, non-**NULL** pointer. For more information about this flag, see the Remarks section.

`ExecutionRoutine`

A pointer to the driver-supplied AdapterListControl routine that initiates the DMA transfer for the driver. The I/O manager calls the *AdapterListControl* routine after the required resources are allocated for the adapter object. After the *AdapterListControl* routine returns, the I/O manager automatically frees the adapter object and the resources that were allocated for this object.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set, the *ExecutionRoutine* parameter is optional and can be NULL. If this parameter is NULL, the caller can use the resources allocated by **GetScatterGatherListEx** to perform the DMA transfer after **GetScatterGatherListEx** returns. For more information, see the Remarks section.

`Context`

The driver-determined, adapter-control context. This context is passed to the AdapterListControl routine as the *Context* parameter.

`WriteToDevice`

The direction of the DMA transfer. Set this parameter to TRUE for a write operation, which transfers data from memory to the device. Set this parameter to FALSE for a read operation, which transfers data from the device to memory.

`DmaCompletionRoutine`

Not used. Set to **NULL**.

`CompletionContext`

Not used. Set to **NULL**.

`*ScatterGatherList`

A pointer to a variable into which the routine writes a pointer to the allocated scatter/gather list. This parameter points to a SCATTER_GATHER_LIST structure. The routine allocates this structure and the **SCATTER_GATHER_ELEMENT** array that it points to.

The *ScatterGatherList* parameter is optional and can be NULL if the *ExecutionRoutine* parameter is non-NULL.

If the **DMA_SYNCHRONOUS_CALLBACK** flag is set and the *ExecutionRoutine* parameter is **NULL**, *ScatterGatherList* must be a valid, non-**NULL** pointer. If *ExecutionRoutine* is non-**NULL**, *ScatterGatherList* is optional and can be **NULL** if the calling driver does not require the scatter/gather list. The **GetScatterGatherListEx** call fails if the **DMA_SYNCHRONOUS_CALLBACK** flag is set and *ScatterGatherList* and *ExecutionRoutine* are both **NULL**, or if the **DMA_SYNCHRONOUS_CALLBACK** flag is not set and *ExecutionRoutine* is **NULL**.

Return Value

GetScatterGatherListEx returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STA TUS _IN VAL ID_ PAR AM ETE RS	The routine failed due to invalid parameter values passed by the caller.
STA TUS _IN SUF FICI ENT _RE SO URC ES	The routine failed to allocate resources required for the DMA transfer.

Remarks

GetScatterGatherListEx
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

Use **GetScatterGatherListEx** only for bus-master adapters. Do not use this routine for a system DMA adapter.

The driver of a bus-master device can use **GetScatterGatherListEx** to combine the operations performed by the AllocateAdapterChannelEx and MapTransferEx routines into a one call. **GetScatterGatherListEx** performs the following operations:

1. Allocates the resources that are required for the DMA transfer.
2. Builds a scatter/gather list based on the values of the *Mdl*, *Offset*, and *Length* parameters.
3. Calls the driver-supplied AdapterListControl routine and supplies the scatter/gather list to this routine as a parameter.

The allocated resources are automatically released after the *AdapterListControl* routine returns. If **GetScatterGatherListEx** is called synchronously (that is, if the **DMA_SYNCHRONOUS_CALLBACK** flag is set), the *AdapterListControl* routine can be omitted. In this case, the caller uses the allocated resources to initiate the DMA transfer after **GetScatterGatherListEx** returns. The caller must explicitly release these resources.

By default, **GetScatterGatherListEx** returns asynchronously, without waiting for the requested resource allocation to complete. After this return, the caller can, if necessary, cancel the pending allocation request by calling the CancelAdapterChannel routine.

If the calling driver sets the **DMA_SYNCHRONOUS_CALLBACK** flag, the **GetScatterGatherListEx** routine behaves as follows:

- If the requested resources are not immediately available, **GetScatterGatherListEx** does not wait for resources, does not build a scatter/gather list, and does not call the *AdapterListControl* routine. Instead, **GetScatterGatherListEx** fails and returns STATUS_INSUFFICIENT_RESOURCES.
- The driver is not required to supply an *AdapterListControl* routine if the **DMA_SYNCHRONOUS_CALLBACK** flag is set.
- If the driver supplies an *AdapterListControl* routine, the **DMA_SYNCHRONOUS_CALLBACK** flag indicates that this routine is to be called in the context of the calling thread, before **GetScatterGatherListEx** returns.
- If the driver does not supply an *AdapterListControl* routine, the driver can use the allocated resources and scatter/gather list after **GetScatterGatherListEx** returns. In this case, the driver must supply a valid, non-NULL *ScatterGatherList* pointer. In addition, after the driver initiates the DMA transfer, the driver must call the FreeAdapterObject routine to free the resources that **GetScatterGatherListEx** allocated for the adapter object.

GetScatterGatherListEx is an extended version of the GetScatterGatherList routine. The following features are available only in the extended version:

GetScatterGatherListEx is similar to the BuildScatterGatherListEx routine, except that **GetScatterGatherListEx** automatically allocates the buffer for the scatter/gather list.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AdapterListControl

AllocateAdapterChannelEx

BuildScatterGatherListEx

CancelAdapterChannel

DEVICE_OBJECT

DMA_ADAPTER

DmaCompletionRoutine

FreeAdapterChannel

GetScatterGatherList

InitializeDmaTransferContext

IoGetDmaAdapter

MapTransferEx

SCATTER_GATHER_LIST

PINITIALIZE_DMA_TRANSFER_CONTEXT callback function

1/25/2019 • 2 minutes to read • Edit Online

The **InitializeDmaTransferContext** routine initializes an opaque DMA transfer context that is used to track pending allocations of DMA resources.

Syntax

```
PINITIALIZE_DMA_TRANSFER_CONTEXT PinitializeDmaTransferContext;

NTSTATUS PinitializeDmaTransferContext(
  PDMA_ADAPTER DmaAdapter,
  PVOID DmaTransferContext
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`DmaTransferContext`

A pointer to a caller-allocated buffer into which **InitializeDmaTransferContext** writes the initial values for the DMA transfer context. This context is opaque to the caller. The caller must allocate a buffer that is large enough to contain the DMA transfer context. The size, in bytes, of this context is specified by the **DMA_TRANSFER_CONTEXT_SIZE_V1** constant in the Wdm.h header file.

Return Value

InitializeDmaTransferContext returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETERS	This routine failed due to invalid parameter values passed by the caller.

Remarks

InitializeDmaTransferContext
is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a
DMA_OPERATIONS*structure*.Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

An initialized DMA transfer context must be supplied as a parameter to the AllocateAdapterChannelEx, GetScatterGatherListEx, or BuildScatterGatherListEx routine. Each of these routines writes information about the requested DMA resource allocation to the DMA transfer context. This information is opaque to the caller. To cancel a pending allocation request, the caller must supply the DMA transfer context for the request to the CancelAdapterChannel routine.

The DMA transfer context that is supplied to **AllocateAdapterChannelEx**, **GetScatterGatherListEx**, or **BuildScatterGatherListEx** must be unique across all adapter allocation requests.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AllocateAdapterChannelEx

BuildScatterGatherListEx

CancelAdapterChannel

DMA_ADAPTER

DMA_OPERATIONS

GetScatterGatherListEx

IoGetDmaAdapter

PINTERFACE_DEREFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceDereference* routine decrements the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_DEREFERENCE PinterfaceDereference;

void PinterfaceDereference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceReference routine to increment the reference count for the interface.

The driver that imports the interface is responsible for calling the *InterfaceDereference* routine to decrement the reference count after the driver is no longer using the interface. For example, a driver that requests a pointer to the interface by sending an IRP_MN_QUERY_INTERFACE request calls *InterfaceDereference*. Also, a driver that receives a pointer to the interface to another driver must call *InterfaceDereference*.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceReference

PINTERFACE_REFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceReference* routine increments the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_REFERENCE PinterfaceReference;

void PinterfaceReference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceDereference routine to decrement the reference count for the interface.

The driver that exports the interface is responsible for calling *InterfaceReference* to increment the reference count before the driver exports the interface. For example, the driver that initially exported the interface in response to an IRP_MN_QUERY_INTERFACE request calls *InterfaceReference*. Also, a driver that passes a pointer to the interface to another driver must call *InterfaceReference*.

Requirements

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

PMAP_TRANSFER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **MapTransfer** routine sets up map registers for an adapter object to map a DMA transfer from a locked-down buffer.

Syntax

```
PMAP_TRANSFER PmapTransfer;

PHYSICAL_ADDRESS PmapTransfer(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  PVOID MapRegisterBase,
  PVOID CurrentVa,
  PULONG Length,
  BOOLEAN WriteToDevice
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA adapter object returned by IoGetDmaAdapter and previously passed to AllocateAdapterChannel for the current IRP's transfer request.

`Mdl`

Pointer to one of the following: the MDL that describes the buffer at **MdlAddress** in the current IRP or the MDL that describes the common buffer set up by the driver of a subordinate device (auto-initialize mode).

`MapRegisterBase`

Specifies the map registers allocated for the DMA operation. The system passes this value to the driver's AdapterControl routine.

`CurrentVa`

Pointer to the current virtual address of the data to be transferred for a DMA transfer operation.

`Length`

Specifies the length, in bytes, to be mapped. If the driver indicated that its device was a bus master with scatter/gather support when it called IoGetDmaAdapter, the value of *Length* on return from **MapTransfer** indicates how many bytes were mapped. Otherwise, the input and output values of *Length* are identical.

`WriteToDevice`

Indicates the direction of the transfer operation: **TRUE** for a transfer from the locked-down buffer to the device.

Return Value

MapTransfer returns the logical address of the region mapped, which the driver of a bus-master adapter can use. Drivers of devices that use a system DMA controller cannot use this value and should ignore it.

Remarks

MapTransfer is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

The *DmaAdapter* must have already been allocated as a result of the driver's preceding call to AllocateAdapterChannel.

The number of map registers that can be set up cannot exceed the maximum returned when the driver called IoGetDmaAdapter.

A driver can get the initial *CurrentVa* for the start of a packet-based DMA transfer by calling MmGetMdlVirtualAddress. However, the value returned is an index into the *Mdl*, rather than a valid virtual address. If the driver must split a large transfer request into more than one DMA operation, it must update *CurrentVa* and *Length* for each DMA operation.

The driver of a bus-master device with scatter/gather support can use the returned logical address and updated *Length* value to build a scatter/gather list, calling **MapTransfer** repeatedly until it has used all available map registers for the transfer operation. However, such a driver could more simply use the GetScatterGatherList routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

ADDRESS_AND_SIZE_TO_SPAN_PAGES

AllocateAdapterChannel

AllocateCommonBuffer

DMA_ADAPTER

FlushAdapterBuffers

FreeAdapterChannel

FreeMapRegisters

IoGetDmaAdapter

KeFlushIoBuffers

MmGetMdlVirtualAddress

PMAP_TRANSFER_EX callback function

1/25/2019 • 7 minutes to read • Edit Online

The **MapTransferEx** routine sets up map registers to map the physical addresses in a scatter/gather list to the logical addresses that are required to do a DMA transfer.

Syntax

```
PMAP_TRANSFER_EX PmapTransferEx;

NTSTATUS PmapTransferEx(
  PDMA_ADAPTER DmaAdapter,
  PMDL Mdl,
  PVOID MapRegisterBase,
  ULONGLONG Offset,
  ULONG DeviceOffset,
  PULONG Length,
  BOOLEAN WriteToDevice,
  PSCATTER_GATHER_LIST ScatterGatherBuffer,
  ULONG ScatterGatherBufferLength,
  PDMA_COMPLETION_ROUTINE DmaCompletionRoutine,
  PVOID CompletionContext
)
{...}
```

Parameters

`DmaAdapter`

A pointer to a DMA_ADAPTER structure. This structure is the adapter object that represents the driver's bus-master DMA device or system DMA channel. The caller obtained this pointer from a previous call to the IoGetDmaAdapter routine.

`Mdl`

A pointer to an MDL chain that describes the physical page layout for a collection of locked-down buffers in virtual memory. The scatter/gather list for the DMA transfer will use the region of this memory that is specified by the *Offset* and *Length* parameters. For more information about MDL chains, see Using MDLs.

`MapRegisterBase`

A handle to the map registers that are allocated for the adapter object. The caller previously obtained this handle from the AllocateAdapterChannelEx routine.

`Offset`

A byte offset from the start of the memory that is described by the MDL chain. This offset specifies the start of the I/O data buffer that is used for the DMA transfer. If a scatter/gather list is supplied to the caller, this offset determines the starting address of the first buffer fragment in the list. If the MDLs in the MDL chain describe a total of N bytes of memory, valid values of *Offset* are in the range 0 to N−1. For more information, see Remarks.

`DeviceOffset`

The byte offset of the target device's data register or FIFO from the device's base address. This parameter applies to devices that have multiple FIFOs that can be accessed by a system DMA controller. This parameter is used only for

system DMA transfers. For bus-master transfers, set this parameter to zero.

`Length`

A pointer to a variable that contains the length, in bytes, of the I/O data buffer that is used for the DMA transfer. On entry, this variable contains the length requested by the calling driver. Before returning, the routine writes the actual length of the mapped buffer to this variable. The value of **Length* on return from **MapTransferEx** indicates how many bytes were mapped. If the number of map registers and the scatter/gather buffer size are sufficient to map the entire length requested by the caller, the input and output values of **Length* are identical. If the MDLs in the MDL chain describe a total of N bytes of memory, valid values of **Length* are in the range 0 to N–*Offset*.

`WriteToDevice`

The direction of the DMA transfer. Set this parameter to **TRUE** for a write operation, which transfers data to the device from memory. Set this parameter to **FALSE** for a read operation, which transfers data from the device to memory.

`ScatterGatherBuffer`

A pointer to a caller-allocated buffer into which the routine writes the scatter/gather list for the DMA transfer. This list begins with a SCATTER_GATHER_LIST structure, which is immediately followed by a **SCATTER_GATHER_ELEMENT** array. For a driver that uses a bus-master DMA device, *ScatterGatherBuffer* is a required parameter. For a driver that uses a system DMA controller, the *ScatterGatherBuffer* parameter is optional and can be **NULL**. For more information, see Remarks.

`ScatterGatherBufferLength`

The size, in bytes, of the buffer that the *ScatterGatherBuffer* parameter points to. The allocated buffer size must be large enough to contain the scatter/gather list, plus internal data that the operating system stores in this buffer. To determine the required buffer size, call the GetDmaTransferInfo or CalculateScatterGatherList routine. If *ScatterGatherBuffer* is **NULL**, set *ScatterGatherBufferLength* to zero.

`DmaCompletionRoutine`

A pointer to a caller-supplied DmaCompletionRoutine routine to be called when the DMA transfer completes. This routine is called if the target device uses a system DMA controller that generates a DMA-completion interrupt. The *DmaCompletionRoutine* routine is called at DISPATCH_LEVEL after the DMA transfer completes. For a system DMA adapter, this parameter is optional and can be **NULL**. For a bus-master adapter, set this parameter to **NULL**.

`CompletionContext`

The driver-determined context for the *DmaCompletionRoutine* routine. This context is supplied as the *CompletionContext* parameter to the *DmaCompletionRoutine* routine. If the *DmaCompletionRoutine* parameter is **NULL**, set *CompletionContext* to **NULL**.

Return Value

MapTransferEx returns STATUS_SUCCESS if the call is successful. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETERS	The routine failed due to invalid parameter values passed by the caller.
STATUS_BUFFER_TOO_SMALL	The caller-supplied buffer in *ScatterGatherBuffer* is too small to contain the scatter/gather list.
STATUS_INSUFFICIENT_RESOURCES	The routine failed to allocate resources required for the DMA transfer.
STATUS_CANCELLED	This transfer was canceled.

Remarks

MapTransferEx *is not a system routine that can be called directly by name. This routine can be called only by pointer from the address returned in a* DMA_OPERATIONS*structure.*Drivers obtain the address of this routine by calling IoGetDmaAdapter with the **Version** member of the *DeviceDescription* parameter set to DEVICE_DESCRIPTION_VERSION3. If **IoGetDmaAdapter** returns **NULL**, the routine is not available on your platform.

For a transfer that uses a system DMA controller, the caller can, as an option, supply a DmaCompletionRoutine callback routine that is called when the transfer finishes. The operating system schedules this callback in response to the DMA completion interrupt from the system DMA controller.

The number of map registers that can be set up by **MapTransferEx** cannot exceed the maximum that the driver obtained from IoGetDmaAdapter.

The *Mdl*, *Offset*, and *Length* parameters describe the I/O data buffer for the requested DMA transfer. The number

of allocated map registers might not be sufficient to map all of the memory in this buffer, or the scatter/gather buffer pointed to by *ScatterGatherBuffer* might not be large enough to describe the entire buffer. **MapTransferEx** writes an output value to **Length* to tell the driver how much of the buffer memory for the requested DMA transfer was mapped by the routine. The routine writes a scatter/gather list to the buffer pointed to by *ScatterGatherBuffer*. This list describes the buffer fragments that were successfully mapped by the routine.

If a call to **MapTransferEx** is successful, **MapTransferEx** writes the **Length* output value before it returns. If the caller specifies a *DmaCompletionRoutine*, the updated **Length* output value is always written before the *DmaCompletionRoutine* runs. For more information, see Multiple calls to MapTransferEx.

The *Offset* parameter specifies the starting offset in the MDL chain that describes the memory in the I/O data buffer. For example, assume that the MDL chain contains two MDLs, MDL_1 and MDL_2, and that MDL_1 describes N_1 bytes of memory, and MDL_2 describes N_2 bytes. If *Offset* = N, where $N_1 < N < N_1 + N_2$, the buffer contains none of the memory described by MDL_1, and starts at an offset of $N - N_1$ bytes in the memory described by MDL_2.

If the transfer uses a system DMA controller, the caller can set *ScatterGatherBuffer* = NULL, in which case **MapTransferEx** uses an internally allocated, default buffer to hold the scatter/gather list. The default buffer is guaranteed to be large enough to contain a scatter/gather list of at least one element. If the default buffer is used for a scatter/gather transfer of many elements, many calls to **MapTransferEx** might be required complete the transfer. If the DMA controller hardware supports scatter/gather transfers, the use of the default buffer might degrade performance.

If *ScatterGatherBuffer* is non-NULL and *ScatterGatherBufferSize* specifies a size that is too small to contain a scatter/gather list of at least one element, **MapTransferEx** fails and returns STATUS_INVALID_PARAMETER.

MapTransferEx is an extended version of the MapTransfer routine. The extended version has these advantages:

- **MapTransferEx** can process all of the buffer fragments in an MDL chain in one call, but **MapTransfer** can process only one physically contiguous buffer fragment per call.
- **MapTransferEx** can generate an entire scatter/gather list in one call, but **MapTransfer** can generate only one scatter/gather list element per call.
- **MapTransferEx** can map all of the buffer fragments in a scatter/gather list in one call, but **MapTransfer** can map only one physically contiguous buffer fragment per call.
- **MapTransferEx** requires only the starting offset for the entire scatter/gather list, but **MapTransfer** requires a starting virtual address for each physically contiguous buffer fragment.
- A **MapTransferEx** call can map a buffer that extends through one or more MDLs, but a **MapTransfer** call can map only one physically contiguous buffer fragment in the memory that is described by an MDL.
- For a system DMA transfer, **MapTransferEx** enables the caller to supply a *DmaCompletionRoutine* callback routine to receive notification after the transfer completes, but **MapTransfer** does not provide a way to notify the caller when a DMA transfer completes.

Each successful call to **MapTransferEx** must be followed by a corresponding call to the FlushAdapterBuffersEx routine. The **FlushAdapterBuffersEx** call that follows a **MapTransferEx** call must occur before the next **MapTransferEx** call occurs. The **FlushAdapterBuffersEx** call is required even if a call to the CancelMappedTransfer routine succeeds in canceling the mapped transfer requested by the preceding **MapTransferEx** call.

For more information, see Using the MapTransferEx Routine.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"

See Also

AllocateAdapterChannelEx

CalculateScatterGatherList

CancelMappedTransfer

DMA_ADAPTER

DMA_OPERATIONS

DmaCompletionRoutine

FlushAdapterBuffersEx

GetDmaTransferInfo

IoGetDmaAdapter

SCATTER_GATHER_LIST

PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentActiveConditionCallback* callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.

Syntax

```
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK PoFxComponentActiveConditionCallback;

void PoFxComponentActiveConditionCallback(
  PVOID Context,
  ULONG Component
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

Specifies the index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Return Value

None.

Remarks

A component's hardware registers can be accessed only when the component is in the active condition. Do not rely on the Fx power state of a component to determine whether the component can be accessed. If the component is in the F0 state and is in the idle condition, the component might be about to switch to another Fx state.

To gain access to a component that is in the idle condition, the driver can call the PoFxActivateComponent routine to switch the component to the active condition. If the component is not already in the active condition when **PoFxActivateComponent** is called, PoFx initiates the transition to the active condition, and then calls the *ComponentActiveConditionCallback* routine to notify the driver when the transition is complete.

The driver receives a *ComponentActiveConditionCallback* callback whenever the component completes a transition to the active condition, regardless of whether the driver called **PoFxActivateComponent** to initiate the transition. For example, the Windows kernel might activate the component on behalf of the driver before sending a Plug and Play IRP or a system set-power IRP to the driver.

Examples

To define a *ComponentActiveConditionCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentActiveConditionCallback* callback routine that is named `MyComponentActiveConditionCallback`, use the PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK MyComponentActiveConditionCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentActiveConditionCallback(
    PVOID Context,
    ULONG Component
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxActivateComponent

PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentCriticalTransitionCallback* callback routine handles a transition of the specified component between the F0 (fully on) and low-power F*x* component power states.

Syntax

```
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK PoFxComponentCriticalTransitionCallback;

void PoFxComponentCriticalTransitionCallback(
  PVOID Context,
  ULONG Component,
  BOOLEAN Active
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. This context is driver-defined and is opaque to PoFx. The driver specified this pointer in the **DeviceContext** member of the PO_FX_CORE_DEVICE structure that the driver used to register the device with the Windows power management framework (PoFx).

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_CORE_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Active`

Indicates whether this notification is for a transition to the F0 component power state. If TRUE, the component has just completed a transition from a low-power F*x* state to F0. If FALSE, the component is about to start a transition from F0 to a low-power F*x* state.

Return Value

None.

Remarks

This callback routine is implemented by a device driver, and is called by PoFx. PoFx calls this routine to notify the driver of power transitions of device components.

The **ComponentCriticalTransitionCallback** member of the PO_FX_CORE_DEVICE structure is a pointer to a *ComponentCriticalTransitionCallback* callback routine. A pointer to a **PO_FX_CORE_DEVICE** structure is passed as an input parameter in the PoFxRegisterCoreDevice call that registers a core device with the Windows power management framework (PoFx).

To handle calls to this routine, the device driver saves or restores the register state or other hardware context of the

specified component in the device.

If *Active* = TRUE, PoFx has called this routine *after* the component completed a transition from a low-power F*x* state to F0. In this call, the routine restores the previously saved hardware context of the component.

If *Active* = FALSE, PoFx has called this routine *before* the component starts a pending transition from F0 to a low-power F*x* state. In this call, the routine saves the component's hardware context so that this context can later be restored.

For more information about F*x* component power states, see Component-Level Power Management.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	wdm.h (include Wudfwdm.h)

See Also

PO_FX_CORE_DEVICE

PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentIdleConditionCallback* callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.

Syntax

```
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK PoFxComponentIdleConditionCallback;

void PoFxComponentIdleConditionCallback(
  PVOID Context,
  ULONG Component
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Return Value

None.

Remarks

When the driver no longer needs to access a component that is in the active condition, the driver should call the PoFxIdleComponent routine to switch the component to the idle condition. In response to the **PoFxIdleComponent** call, PoFx initiates the transition to the idle condition, and then calls the *ComponentIdleConditionCallback* routine to notify the driver when this transition is complete.

The driver must call **PoFxCompleteIdleCondition** in response to every *ComponentIdleConditionCallback* callback. The **PoFxCompleteIdleCondition** call can occur either during the *ComponentIdleConditionCallback* callback or after the callback returns. The driver should complete any work that requires access to the hardware component before the call to **PoFxCompleteIdleCondition**. After this call, the component might not stay in the F0 power state.

A component can be safely accessed only when the component is in the active condition. Do not rely on the Fx power state of a component to determine whether the component can be accessed. If the component is in the F0

state and is in the idle condition, the component might be about to switch to another Fx state.

Examples

To define a *ComponentIdleConditionCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentIdleConditionCallback* callback routine that is named `MyComponentIdleConditionCallback`, use the PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK MyComponentIdleConditionCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentIdleConditionCallback(
    PVOID Context,
    ULONG Component
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxCompleteIdleCondition

PO_FX_COMPONENT_IDLE_STATE_CALLBACK callback function

1/25/2019 • 4 minutes to read • Edit Online

The *ComponentIdleStateCallback* callback routine notifies the driver of a pending change to the Fx power state of the specified component.

Syntax

```
PO_FX_COMPONENT_IDLE_STATE_CALLBACK PoFxComponentIdleStateCallback;

void PoFxComponentIdleStateCallback(
  PVOID Context,
  ULONG Component,
  ULONG State
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

Specifies the component number. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N-1.

`State`

Specifies the new Fx power state that the component will change to. If this parameter is zero, the new state is F0; if this parameter is one, the new state is F1; and so on.

Return Value

None.

Remarks

When PoFx calls the driver's *ComponentIdleStateCallback* routine, the driver might need to prepare for the pending Fx state change. After any necessary preparations are completed, the driver must call the PoFxCompleteIdleState routine to notify PoFx that the driver completed its response to the *ComponentIdleStateCallback* callback. The **PoFxCompleteIdleState** call can occur either before or after the *ComponentIdleStateCallback* routine returns.

If the component is to switch from F0 to a low-power Fx state in which the device will lose the hardware state of

the component, the driver must save the component's hardware state before the transition to the new Fx state occurs. If the component is to switch from a low-power Fx state to F0, and the hardware state was previously saved, the driver should restore the hardware state after power is restored to the component.

For some devices, the power state of a component might be controlled in the device hardware. For these devices, the driver is typically responsible for changing the power state of the component. For other devices, the power state of a component might be controlled by hardware that is external to the device, and the driver typically must rely on PoFx to configure the power state of the component.

If the driver is responsible for configuring the power state of the component, the driver should change the power state in response to the *ComponentIdleStateCallback* callback. For a transition from F0 to a low-power Fx state, the component is in F0 on entry to the *ComponentIdleStateCallback* routine, and the driver must switch the component the new Fx state before calling **PoFxCompleteIdleState**. For a transition from a low-power Fx state to F0, the component is in the low-power Fx state on entry to the *ComponentIdleStateCallback* routine, and the driver must switch the component to F0 before calling **PoFxCompleteIdleState**.

If the driver is not responsible for configuring the power state of the component, the driver should assume that the component is always in the F0 state on entry to the *ComponentIdleStateCallback* routine. For a pending transition from F0 to a low-power Fx state, the transition to the new Fx state does not occur until after the driver calls **PoFxCompleteIdleState**. For a transition from a low-power Fx state to F0, the transition to F0 occurs before the *ComponentIdleStateCallback* routine is called.

PoFx never calls the *ComponentIdleStateCallback* routine to switch directly from one low-power Fx state to another low-power Fx state. For example, PoFx might need to switch a component from one low-power Fx state to another in response to a call to the PoFxSetComponentLatency, PoFxSetComponentResidency, or PoFxSetComponentWake routine. In this case, PoFx first calls the *ComponentIdleStateCallback* routine to switch from the old Fx state to F0, and then calls the *ComponentIdleStateCallback* routine a second time to switch from F0 to the new Fx state.

Examples

To define a *ComponentIdleStateCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentIdleStateCallback* callback routine that is named `MyComponentIdleStateCallback`, use the PO_FX_COMPONENT_IDLE_STATE_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_IDLE_STATE_CALLBACK MyComponentIdleStateCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentIdleStateCallback(
    PVOID Context,
    ULONG Component,
    ULONG State
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_IDLE_STATE_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_IDLE_STATE_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows 8 and later versions of Windows.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxCompleteIdleState

PoFxSetComponentLatency

PoFxSetComponentResidency

PoFxSetComponentWake

PO_FX_COMPONENT_PERF_STATE_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentPerfStateCallback* callback routine notifies the driver that its request to change the performance state of a component is complete.

Syntax

```
PO_FX_COMPONENT_PERF_STATE_CALLBACK PoFxComponentPerfStateCallback;

void PoFxComponentPerfStateCallback(
  PVOID Context,
  ULONG Component,
  BOOLEAN Succeeded,
  PVOID RequestContext
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

Specifies the index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Succeeded`

If TRUE, the platform extension plug-in (PEP) succeeded the performance state change. If FALSE, the PEP failed the performance state change. For more information, see the Remarks section.

`RequestContext`

Pointer to the optional driver or device context that was specified by the *Context* parameter of the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.

Return Value

None.

Remarks

When a device driver requests a performance state change by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine, PoFx calls the *ComponentPerfStateCallback* routine to

notify the driver when the request is complete.

To specify a *ComponentPerfStateCallback* routine, use the *ComponentPerfStateCallback* parameter of the PoFxRegisterComponentPerfStates routine when the driver registers for performance state management by PoFx. The *ComponentPerfStateCallback* routine may be the same for all components and all devices.

If *Succeeded* is TRUE, the driver should perform whatever work is necessary to change the performance state in the hardware. If *Succeeded* is FALSE, the driver may choose to do nothing or retry the request with the same performance state or an alternate performance state.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Desktop
Header	wdm.h
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PoFxIssueComponentPerfStateChange

PoFxIssueComponentPerfStateChangeMultiple

PoFxRegisterDevice

PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DevicePowerNotRequiredCallback* callback routine notifies the device driver that the device is not required to stay in the D0 power state.

Syntax

```
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK PoFxDevicePowerNotRequiredCallback;

void PoFxDevicePowerNotRequiredCallback(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

Return Value

None.

Remarks

When PoFx calls the driver's *DevicePowerNotRequiredCallback* routine, the driver must first decide whether to initiate a transition to a low-power Dx state (by sending an IRP_MN_SET_POWER request down the device stack) or to remain in the D0 state. Next, without waiting for any Dx transition to complete, the driver must call the PoFxCompleteDevicePowerNotRequired routine to notify PoFx that the driver completed its response to the *DevicePowerNotRequiredCallback* callback. The driver can call **PoFxCompleteDevicePowerNotRequired** before or after the *DevicePowerNotRequiredCallback* routine returns.

When a device is in the D0 state, and the Fx state or active/idle condition of a component in the device changes, PoFx evaluates whether the device can enter a low-power Dx state or must remain in the D0 state. If the device can enter a low-power Dx state, PoFx calls the driver's *DevicePowerNotRequiredCallback* routine.

If the device enters a low-power Dx state in response to a *DevicePowerNotRequiredCallback* callback, but PoFx later determines that the device must enter the D0 state, PoFx calls the driver's DevicePowerRequiredCallback routine. In response to this call, the device must enter the D0 state.

Examples

To define a *DevicePowerNotRequiredCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV),

and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DevicePowerNotRequiredCallback* callback routine that is named `MyDevicePowerNotRequiredCallback`, use the PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK type as shown in this code example:

```
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK MyDevicePowerNotRequiredCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDevicePowerNotRequiredCallback(
    PVOID Context
    )
  {
      // Function body
  }
```

The PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

DevicePowerRequiredCallback

IRP_MN_SET_POWER

PO_FX_DEVICE

PoFxCompleteDevicePowerNotRequired

PoFxRegisterDevice

PO_FX_DEVICE_POWER_REQUIRED_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DevicePowerRequiredCallback* callback routine notifies the device driver that the device must enter and remain in the D0 power state.

Syntax

```
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK PoFxDevicePowerRequiredCallback;

void PoFxDevicePowerRequiredCallback(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

Return Value

None.

Remarks

After PoFx calls the driver's *DevicePowerRequiredCallback* routine, the driver must initiate a transition to the D0 state by sending a D0 IRP (an IRP_MN_SET_POWER request) down the device stack, if the device is not already in the D0 state. After the driver completes (and all lower drivers complete) handling of this request, the driver must call the PoFxReportDevicePoweredOn routine. The **PoFxReportDevicePoweredOn** call notifies PoFx that the device completed its response to the *DevicePowerRequiredCallback* call. The driver can call **PoFxReportDevicePoweredOn** during or after the *DevicePowerRequiredCallback* call.

Each time the *DevicePowerRequiredCallback* routine is called, the driver must respond by calling **PoFxReportDevicePoweredOn**. The **PoFxReportDevicePoweredOn** call can occur either during or after the *DevicePowerRequiredCallback* call. The driver must call **PoFxReportDevicePoweredOn** regardless of whether no D0 IRP is required, the D0 IRP request fails, or the dispatched D0 IRP succeeds or fails.

PoFx calls the *DevicePowerRequiredCallback* routine only if PoFx previously called the driver's DevicePowerNotRequiredCallback routine to notify the driver that the device was not required to stay in the D0 state.

The call to the *DevicePowerRequiredCallback* routine can occur at any time after the driver calls PoFxCompleteDevicePowerNotRequired to complete a *DevicePowerNotRequiredCallback* callback. The *DevicePowerRequiredCallback* call does not necessarily occur as a result of an action taken by the driver, such as

requesting that a component change to the active condition.

Examples

To define a *DevicePowerRequiredCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DevicePowerRequiredCallback* callback routine that is named `MyDevicePowerRequiredCallback`, use the PO_FX_DEVICE_POWER_REQUIRED_CALLBACK type as shown in this code example:

```
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK MyDevicePowerRequiredCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDevicePowerRequiredCallback(
    PVOID Context
    )
  {
      // Function body
  }
```

The PO_FX_DEVICE_POWER_REQUIRED_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_DEVICE_POWER_REQUIRED_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

DevicePowerNotRequiredCallback

PO_FX_DEVICE

PoFxCompleteDevicePowerNotRequired

PoFxRegisterDevice

PoFxReportDevicePoweredOn

PO_FX_POWER_CONTROL_CALLBACK callback function

1/25/2019 • 3 minutes to read • Edit Online

The *PowerControlCallback* callback routine performs a power control operation that is requested by the power management framework (PoFx).

Syntax

```
PO_FX_POWER_CONTROL_CALLBACK PoFxPowerControlCallback;

NTSTATUS PoFxPowerControlCallback(
  PVOID DeviceContext,
  LPCGUID PowerControlCode,
  PVOID InBuffer,
  SIZE_T InBufferSize,
  PVOID OutBuffer,
  SIZE_T OutBufferSize,
  PSIZE_T BytesReturned
)
{...}
```

Parameters

`DeviceContext`

`PowerControlCode`

A pointer to the power control code. This code is a GUID value that specifies the requested operation.

`InBuffer`

A pointer to the buffer that contains the input data for the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *InBuffer* parameter is optional and can be specified as NULL if the specified operation requires no input data.

`InBufferSize`

The size, in bytes, of the input buffer that is pointed to by the *InBuffer* parameter. If *InBuffer* is NULL, this parameter is zero.

`OutBuffer`

A pointer to the buffer to which the callback routine writes the output data from the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *OutBuffer* parameter is optional and can be specified as NULL if the specified operation produces no output data.

`OutBufferSize`

The size, in bytes, of the output buffer that is pointed to by the *OutBuffer* parameter. If *OutBuffer* is NULL, this parameter is zero.

`BytesReturned`

A pointer to a location into which the routine writes the number of bytes of data that were written to the buffer that is pointed to by *OutBuffer*. The number of bytes written must be less than or equal to *OutBufferSize*. This parameter is optional and can be specified as NULL if the caller does not need to know how many bytes were written to the output buffer.

Return Value

The *PowerControlCallback* routine returns STATUS_SUCCESS if the call is successful. Otherwise, it returns an appropriate error code.

Remarks

PoFx calls this routine to send a power control request directly to the device driver. A power control request is similar to an I/O control request (IOCTL). Unlike an IOCTL, however, a power control request is sent directly to the driver and is not observed by other device drivers in the device stack. During a *PowerControlCallback* call, the driver synchronously performs the requested operation.

This routine is optional. A device driver that does not support power control operations is not required to implement a *PowerControlCallback* routine.

The device driver can call the PoFxPowerControl routine to send a power control request to PoFx.

For more information about power control requests, see PoFxPowerControl.

Examples

To define a *PowerControlCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *PowerControlCallback* callback routine that is named `MyPowerControlCallback`, use the PO_FX_POWER_CONTROL_CALLBACK type as shown in this code example:

```
PO_FX_POWER_CONTROL_CALLBACK MyPowerControlCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyPowerControlCallback(
    PVOID Context,
    LPCGUID PowerControlCode,
    PVOID InBuffer,
    SIZE_T InBufferSize,
    PVOID OutBuffer,
    SIZE_T OutBufferSize,
    PSIZE_T BytesReturned
    )
  {
      // Function body
  }
```

The PO_FX_POWER_CONTROL_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_POWER_CONTROL_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxPowerControl

POB_POST_OPERATION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ObjectPostCallback* routine is called by the operating system after a process or thread handle operation occurs.

Syntax

```
POB_POST_OPERATION_CALLBACK PobPostOperationCallback;

void PobPostOperationCallback(
  PVOID RegistrationContext,
  POB_POST_OPERATION_INFORMATION OperationInformation
)
{...}
```

Parameters

`RegistrationContext`

The context that the driver specifies as the *CallBackRegistration*->**RegistrationContext** parameter of the ObRegisterCallbacks routine. The meaning of this value is driver-defined.

`OperationInformation`

A pointer to an OB_POST_OPERATION_INFORMATION structure that specifies the parameters of the handle operation.

Return Value

None

Remarks

Use the ObRegisterCallbacks routine to register an *ObjectPostCallback* routine, and use the ObUnRegisterCallbacks routine to unregister the routine.

An *ObjectPostCallback* routine is called after an operation on a process handle or thread handle completes, whereas an ObjectPreCallback routine is called before the operation occurs. The *OperationInformation* parameter of the *ObjectPreCallback* routine is a pointer to an OB_PRE_OPERATION_INFORMATION structure, which contains information that you can modify. In contrast, the *OperationInformation* parameter of the *ObjectPostCallback* routine is a pointer to an OB_POST_OPERATION_INFORMATION structure. The contents of this structure are informational and you cannot modify them.

This routine is called at PASSIVE_LEVEL in an arbitrary thread context with normal kernel APCs disabled. Special kernel APCs are not disabled. For more information about APCs, see Types of APCs.

Requirements

Minimum supported client	Available starting with Windows Server 2008.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL (see Remarks section).

See Also

OB_POST_OPERATION_INFORMATION

ObRegisterCallbacks

ObUnRegisterCallbacks

ObjectPreCallback

POB_PRE_OPERATION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ObjectPreCallback* routine is called by the operating system when a process or thread handle operation occurs.

Syntax

```
POB_PRE_OPERATION_CALLBACK PobPreOperationCallback;

OB_PREOP_CALLBACK_STATUS PobPreOperationCallback(
  PVOID RegistrationContext,
  POB_PRE_OPERATION_INFORMATION OperationInformation
)
{...}
```

Parameters

`RegistrationContext`

The context that the driver specifies as the *CallBackRegistration*->**RegistrationContext** parameter of the ObRegisterCallbacks routine. The meaning of this value is driver-defined.

`OperationInformation`

A pointer to an OB_PRE_OPERATION_INFORMATION structure that specifies the parameters of the handle operation.

Return Value

ObjectPreCallback returns an OB_PREOP_CALLBACK_STATUS value. Drivers must return OB_PREOP_SUCCESS.

Remarks

Use the ObRegisterCallbacks routine to register an *ObjectPreCallback* routine, and use the ObUnRegisterCallbacks routine to unregister the routine.

This routine is called at PASSIVE_LEVEL in an arbitrary thread context with normal kernel APCs disabled. Special kernel APCs are not disabled. For more information about APCs, see Types of APCs.

Requirements

Minimum supported client	Available starting with Windows Server 2008.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL (see Remarks section).

See Also

OB_PRE_OPERATION_INFORMATION

ObRegisterCallbacks

ObUnRegisterCallbacks

ObjectPostCallback

PoCallDriver function

1/25/2019 • 2 minutes to read • Edit Online

The **PoCallDriver** routine passes a power IRP to the next-lower driver in the device stack. (Windows Server 2003, Windows XP, and Windows 2000 only.)

Syntax

```
NTSTATUS PoCallDriver(
  PDEVICE_OBJECT        DeviceObject,
  __drv_aliasesMem PIRP Irp
);
```

Parameters

`DeviceObject`

A pointer to the driver-created DEVICE_OBJECT to which the IRP is to be routed.

`Irp`

A pointer to an IRP.

Return Value

PoCallDriver returns STATUS_SUCCESS to indicate success. It returns STATUS_PENDING if it has queued the IRP.

Remarks

Beginning with Windows Vista, drivers should call IoCallDriver, not **PoCallDriver** to pass a power IRP to the next-lower driver. However, on Windows Server 2003, Windows XP, and Windows 2000, drivers must call **PoCallDriver**, not **IoCallDriver** to pass a power IRP to the next-lower driver. On Windows Server 2003, Windows XP, an Windows 2000, drivers must also call PoStartNextPowerIrp before calling **PoCallDriver**.

A driver that requires a new IRP should call PoRequestPowerIrp. A driver must not allocate its own power IRP.

When passing a power IRP down to the next-lower driver, the caller should use IoSkipCurrentIrpStackLocation or IoCopyCurrentIrpStackLocationToNext to set the IRP stack location, then call **PoCallDriver**. Use **IoCopyCurrentIrpStackLocationToNext** if processing the IRP requires setting an IoCompletion routine, or **IoSkipCurrentStackLocation** if no *IoCompletion* routine is required.

When a device is powering up, its drivers must set *IoCompletion* routines to perform start-up tasks (initializing the device, restoring context, and so on) after the bus driver has set the device in the working state. Set *IoCompletion* routines before calling **PoCallDriver**.

When a device is powering down, its drivers must perform necessary power-down tasks before passing the IRP to the next lower driver. After the IRP has reached the bus driver, the device will be powered off and its drivers will no longer have access to it. On Windows Server 2003, Windows XP, and Windows 2000, an *IoCompletion* routine that is associated with a power-down IRP is only required to call **PoStartNextPowerIrp**.

Only one inrush IRP can be active in the system at a time. When passing a power-up IRP for a device that requires

inrush current (in other words, the DO_POWER_INRUSH flag is set in the device object), **PoCallDriver** checks whether another inrush IRP is already active. If so, **PoCallDriver** queues the current IRP for handling after the previous IRP completes, and then returns STATUS_PENDING. For more information about inrush IRPs, see Setting Device Object Flags for Power Management.

If an IRP_MN_SET_POWER or IRP_MN_QUERY_POWER request is already active for *DeviceObject*, **PoCallDriver** queues this IRP and returns STATUS_PENDING.

On Windows 2000 and later systems, pageable drivers (the DO_POWER_PAGABLE flag is set in the device object) must call **PoCallDriver** at IRQL = PASSIVE_LEVEL. Drivers that cannot be paged (DO_POWER_PAGABLE is not set in the device object) or that require inrush current (DO_POWER_INRUSH is set in the device object) can call **PoCallDriver** at IRQL = PASSIVE_LEVEL or DISPATCH_LEVEL.

On Windows 98/Me, all drivers call **PoCallDriver** at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.
DDI compliance rules	CompleteRequestStatusCheck, CompletionRoutineRegistered, DeleteDevice, ForwardedAtBadIrql, ForwardedAtBadIrqlAllocate, ForwardedAtBadIrqlFsdAsync, ForwardedAtBadIrqlFsdSync, IoAllocateForward, IoAllocateIrpSignalEventInCompletionTimeout, IoBuildDeviceControlWait, IoBuildDeviceControlWaitTimeout, IoBuildFsdForward, IoBuildSynchronousFsdRequestWait, IoBuildSynchronousFsdRequestWaitTimeout, IoSetCompletionRoutineExCheck, IrpProcessingComplete, LowerDriverReturn, MarkDevicePower, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, MarkPower, MarkPowerDown, MarkQueryRelations, MarkStartDevice, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, PnpIrpCompletion, PowerDownFail, PowerUpFail, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove2, RemoveLockMnSurpriseRemove, RemoveLockQueryMnRemove, TargetRelationNeedsRef, WmiForward, HwStorPortProhibitedDDIs

See Also

IRP

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IoCallDriver

IoCompletion

IoCopyCurrentIrpStackLocationToNext

IoSkipCurrentIrpStackLocation

PoRequestPowerIrp

PoStartNextPowerIrp

PoClearPowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoClearPowerRequest** routine decrements the count for the specified power request type.

Syntax

```
NTSTATUS PoClearPowerRequest(
  PVOID              PowerRequest,
  POWER_REQUEST_TYPE Type
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

`Type`

The type of the power request. Set this parameter to the following POWER_REQUEST_TYPE enumeration value:

- **PowerRequestSystemRequired**

Return Value

PoClearPowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_UNSUPPORTED	Parameter *Type* is set to an unsupported value.

Remarks

A driver can call the PoSetPowerRequest routine to request that the power manager override several types of default power behavior, which are specified as POWER_REQUEST_TYPE enumeration values. To restore the default behavior, the driver cancels the request by calling the **PoClearPowerRequest** routine.

The power manager maintains a count of the active requests for each power request type. The **PoSetPowerRequest** routine increments the count for the specified power request type by one. The **PoClearPowerRequest** routine decrements the count by one. A nonzero count indicates that requests from one or more components are active. After the count decrements to zero, the power manager restores the default behavior for the specified power request type.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

POWER_REQUEST_TYPE

PoCreatePowerRequest

PoSetPowerRequest

PoCreatePowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoCreatePowerRequest** routine creates a power request object.

Syntax

```
NTSTATUS PoCreatePowerRequest(
  PVOID                    *PowerRequest,
  PDEVICE_OBJECT           DeviceObject,
  PCOUNTED_REASON_CONTEXT Context
);
```

Parameters

`PowerRequest`

A pointer to a location into which the routine writes a pointer to the newly created power request object. If the call fails, the routine writes **NULL** to this location.

`DeviceObject`

A pointer to the device object of the caller (a DEVICE_OBJECT structure).

`Context`

A pointer to a COUNTED_REASON_CONTEXT structure that describes why the caller is creating the power request object. This parameter is optional and can be set to **NULL**.

Return Value

PoCreatePowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *DeviceObject* parameter is **NULL**.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	There is not enough memory available to create a power request object.

Remarks

This routine creates a power request object. To enable power requests, the caller should create one power request object and use that object for all calls to the PoSetPowerRequest and PoClearPowerRequest routines.

A driver can use power requests to override certain aspects of the computer's default power behavior. For example, a driver for a TV receiver device can use power requests to prevent the power manager from automatically blanking the display during extended periods of time in which no user interaction occurs.

When the power request object is no longer needed, the caller must delete the object by calling the PoDeletePowerRequest routine. The driver must delete the power request object before it deletes the device object that was used to create the power request object.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

PoClearPowerRequest

PoDeletePowerRequest

PoSetPowerRequest

PoDeletePowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoDeletePowerRequest** routine deletes a power request object.

Syntax

```
void PoDeletePowerRequest(
  PVOID PowerRequest
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

Return Value

None

Remarks

The driver must delete the power request object before it deletes the device object that was used to create the power request object.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoCreatePowerRequest

PoEndDeviceBusy function

1/25/2019 • 2 minutes to read • Edit Online

The **PoEndDeviceBusy** routine marks the end of a period of time in which the device is busy.

Syntax

```
void PoEndDeviceBusy(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoEndDeviceBusy**.

Return Value

None

Remarks

The PoStartDeviceBusy and **PoEndDeviceBusy** routines mark the start and end of a time period in which a device is busy. Each call to **PoStartDeviceBusy** must be followed by a corresponding call to **PoEndDeviceBusy**. For more information about how these routines work together, see PoStartDeviceBusy.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

PoRegisterDeviceForIdleDetection

PoStartDeviceBusy

PoFxActivateComponent function

1/25/2019 • 4 minutes to read • Edit Online

The **PoFxActivateComponent** routine increments the activation reference count on the specified component.

Syntax

```
void PoFxActivateComponent(
  POHANDLE Handle,
  ULONG    Component,
  ULONG    Flags
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Flags`

The flags for the activation operation. Set this member to zero or to one of the following flag **PO_FX_FLAG_*XXX*** bits:

These two flag bits are mutually exclusive. For more information, see Remarks.

VALUE	MEANING
PO_FX_FLAG_BLOCKING 0x1	Make the condition change synchronous. If this flag is set, the routine that requests the condition change does not return control to the calling driver until the component hardware completes the transition to the new condition. This flag can be used only if the caller is running at IRQL < DISPATCH_LEVEL.

PO_FX_FLAG_ASYNC_ONLY 0x2	Make the condition change fully asynchronous. If this flag is set, the calling driver's callback routine is called from a thread other than the thread in which the routine that requests the condition change is called. Thus, the routine that requests the condition change always returns asynchronously without waiting for the callback to complete.

Return Value

None.

Remarks

Before a device driver can access a component in a device, the driver must first call **PoFxActivateComponent** to obtain an activation reference to the component. If the component is not already in the active condition, this call initiates a transition from the idle condition to the active condition. When this transition completes, PoFx calls the driver's ComponentActiveConditionCallback routine to notify the driver. The driver can access the hardware registers in a component only when the component is in the active condition.

If the component is already in the active condition when **PoFxActivateComponent** is called, no transition is required, and the *ComponentActiveConditionCallback* routine is not called.

After a component enters the active condition, it remains in the active condition for as long as the driver holds one or more activation references on the component. To release an activation reference, the driver calls the PoFxIdleComponent routine. When the driver releases the last activation reference on a component, **PoFxIdleComponent** initiates a transition from the active condition to the idle condition. A component that is in the idle condition can potentially enter a low-power Fx state.

If *Flags* = **PO_FX_FLAG_BLOCKING**, the **PoFxActivateComponent** call is synchronous. If the component is already in the active condition, the call increments the activation reference count and returns without waiting. Otherwise, **PoFxActivateComponent** waits to return until the component completes the transition to the active condition. In this case, if the component is not already in the F0 state when the call occurs, **PoFxActivateComponent** calls the driver's *ComponentIdleStateCallback* routine to initiate the transition to F0. After the component enters the F0 state, **PoFxActivateComponent** calls the driver's *ComponentActiveConditionCallback* routine to inform the driver that the component is in the active condition. These callbacks occur in the same thread as the call to **PoFxActivateComponent**, and **PoFxActivateComponent** returns only after the *ComponentActiveConditionCallback* callback returns.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY**, the **PoFxActivateComponent** call is asynchronous. If the component is already in the active condition, the call increments the activation reference count and returns. Otherwise, **PoFxActivateComponent** schedules the *ComponentIdleStateCallback* (if necessary) and *ComponentActiveConditionCallback* callbacks to occur in another thread, and then returns without waiting for either callback to occur. The callbacks can occur before or after **PoFxActivateComponent** returns. The driver relies on the *ComponentActiveConditionCallback* callback to determine when the component completes the transition to the active condition.

The driver can set *Flags* = 0 to indicate that it does not care whether the **PoFxActivateComponent** call is synchronous or asynchronous. In this case, PoFx decides whether to make the call synchronous or asynchronous.

Two or more code paths in the same driver might need to concurrently access a particular component. The **PoFxActivateComponent** and **PoFxIdleComponent** routines use activation reference counts to enable the various parts of the driver to independently maintain access to the component without requiring the driver to

centrally manage access to the component.

PoFx maintains an activation reference count for each component in a device. A **PoFxActivateComponent** call increments this count by one, and a **PoFxIdleComponent** call decrements the count by one. When the count is nonzero, the component either is in the active condition or is in the process of switching to the active condition. A component that has a count of zero either is in the idle condition or is in the process of switching to the idle condition.

When a **PoFxActivateComponent** call causes the activation reference count to increment from 0 to 1, **PoFxActivateComponent** initiates a transition from the idle condition to the active condition. When a **PoFxIdleComponent** call causes the count to decrement from 1 to 0, **PoFxIdleComponent** initiates a transition from the active condition to the idle condition.

PoFx notifies the driver when a transition between the active condition and idle condition occurs. A *ComponentActiveConditionCallback* callback notifies the driver of a transition to the active condition, and a ComponentIdleConditionCallback callback notifies the driver of a transition to the idle condition. When a **PoFxActivateComponent** or **PoFxIdleComponent** call simply increments or decrements the activation reference count without causing such a transition, the driver receives no notification.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ComponentActiveConditionCallback

ComponentIdleStateCallback

PO_FX_DEVICE

PoFxIdleComponent

PoFxRegisterDevice

PoFxCompleteDevicePowerNotRequired function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxCompleteDevicePowerNotRequired** routine notifies the power management framework (PoFx) that the calling driver has completed its response to a call to the driver's DevicePowerNotRequiredCallback callback routine.

Syntax

```
void PoFxCompleteDevicePowerNotRequired(
  POHANDLE Handle
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

Return Value

None.

Remarks

After PoFx calls the driver's DevicePowerNotRequiredCallback callback routine, the driver must respond by calling **PoFxCompleteDevicePowerNotRequired**.

PoFx calls the driver's *DevicePowerNotRequiredCallback* routine to inform the driver that the device is no longer required to remain in the D0 state. In response to this call, the driver can choose to enter a low-power Dx state or to remain in the D0 state. After the driver has completed its response to the *DevicePowerNotRequiredCallback* call, the driver calls **PoFxCompleteDevicePowerNotRequired** to inform PoFx. The **PoFxCompleteDevicePowerNotRequired** call can occur either during or after the *DevicePowerNotRequiredCallback* call.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe

IRQL	"<= DISPATCH_LEVEL"

See Also

DevicePowerNotRequiredCallback

PoFxRegisterDevice

PoFxCompleteIdleCondition function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxCompleteIdleCondition** routine informs the power management framework (PoFx) that the specified component has completed a pending change to the idle condition.

Syntax

```
void PoFxCompleteIdleCondition(
  POHANDLE Handle,
  ULONG    Component
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N−1.

Return Value

None.

Remarks

After PoFx calls the driver's ComponentIdleConditionCallback callback routine, the driver must respond by calling **PoFxCompleteIdleCondition**.

When PoFx calls the driver's *ComponentIdleConditionCallback* routine, the driver initiates a transition to the idle condition in the specified component. After this transition is complete, the driver calls **PoFxCompleteIdleCondition** to notify PoFx. The **PoFxCompleteIdleCondition** call can occur either during or after the *ComponentIdleConditionCallback* call.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h

Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ComponentIdleConditionCallback

PO_FX_DEVICE

PoFxRegisterDevice

PoFxCompleteIdleState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxCompleteIdleState** routine informs the power management framework (PoFx) that the specified component has completed a pending change to an Fx state.

Syntax

```
void PoFxCompleteIdleState(
  POHANDLE Handle,
  ULONG    Component
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N−1.

Return Value

None,

Remarks

After PoFx calls the driver's ComponentIdleStateCallback callback routine, the driver must respond by calling **PoFxCompleteIdleState**.

PoFx calls the driver's *ComponentIdleStateCallback* routine to tell the driver to switch the specified component to a new Fx power state. After the driver completes the transition to the new Fx state, the driver calls **PoFxCompleteIdleState** to inform PoFx. The **PoFxCompleteIdleState** call can occur either during or after the *ComponentIdleStateCallback* call.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h

Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ComponentIdleStateCallback

PO_FX_DEVICE

PoFxRegisterDevice

PoFxIdleComponent function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxIdleComponent** routine decrements the activation reference count on the specified component.

Syntax

```
void PoFxIdleComponent(
  POHANDLE Handle,
  ULONG    Component,
  ULONG    Flags
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Flags`

The flags for the idle operation. Set this member to zero or to one of the following flag PO_FX_FLAG_XXX bits:

- **PO_FX_FLAG_BLOCKING**
- **PO_FX_FLAG_ASYNC_ONLY**

These two flag bits are mutually exclusive. For more information, see Remarks.

Return Value

None.

Remarks

A device driver calls **PoFxIdleComponent** to release an activation reference to a component in a device. The driver obtained the activation reference in a previous call to the PoFxActivateComponent routine. The driver should hold an activation reference on a component only while the driver needs to access the component. To hold an activation reference on a component that is not being used prevents the component from entering a low-power Fx state.

If the driver holds no other activation references to the component, **PoFxIdleComponent** initiates a transition from the active condition to the idle condition. When this transition completes, PoFx calls the driver's ComponentIdleConditionCallback routine to notify the driver. If the driver retains one or more additional activation references on the component, the component stays in the active condition, and the *ComponentIdleConditionCallback* routine is not called.

PoFx maintains an activation reference count for each component in the device. The **PoFxActivateComponent** routine increments this count, and **PoFxIdleComponent** decrements it. The component remains in the active condition while this count is nonzero. When the driver releases its last activation reference to a component, the count decrements to zero and the component enters the idle condition. After the component enters the idle condition, PoFx can potentially switch the component to a low-power Fx state. For more information, see PoFxActivateComponent.

If *Flags* = **PO_FX_FLAG_BLOCKING**, the **PoFxIdleComponent** call is synchronous. In this case, **PoFxIdleComponent** waits to return until the component completes the transition to the idle condition. **PoFxIdleComponent** calls the driver's *ComponentIdleConditionCallback* callback routine to inform the driver that the component is in the idle condition. This callback occurs in the same thread as the call to **PoFxIdleComponent**, and **PoFxIdleComponent** returns only after the *ComponentIdleConditionCallback* callback returns.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY**, the **PoFxIdleComponent** call is asynchronous. In this case, **PoFxIdleComponent** schedules the *ComponentIdleConditionCallback* callback to occur in another thread, and then returns without waiting for the callback to occur. The callback can occur before or after **PoFxIdleComponent** returns. The driver should rely on the *ComponentIdleConditionCallback* callback to determine when the component completes the transition to the idle condition. Until this callback occurs, the driver should assume that the component might still be in the active condition.

The driver can set *Flags* = 0 to indicate that it does not care whether the **PoFxIdleComponent** call is synchronous or asynchronous. In this case, PoFx decides whether to make the call synchronous or asynchronous.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ComponentIdleConditionCallback

PO_FX_DEVICE

PO_FX_FLAG_XXX

PoFxActivateComponent

PoFxRegisterDevice

PoFxIssueComponentPerfStateChange function

1/25/2019 • 3 minutes to read • Edit Online

The **PoFxIssueComponentPerfStateChange** routine submits a request to place a device component in a particular performance state.

Syntax

```
void PoFxIssueComponentPerfStateChange(
  POHANDLE                 Handle,
  ULONG                    Flags,
  ULONG                    Component,
  PPO_FX_PERF_STATE_CHANGE PerfChange,
  PVOID                    Context
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`Flags`

The flags that modify the behavior of the performance state change operation. Set this member to zero or to one of the following flag **PO_FX_FLAG_*XXX*** bits:

These two flag bits are mutually exclusive. For more information, see Remarks.

VALUE	MEANING
PO_FX_FLAG_BLOCKING 0x1	Make the condition change synchronous. If this flag is set, the routine that requests the condition change does not return control to the calling driver until the component hardware completes the transition to the new condition. This flag can be used only if the caller is running at IRQL < DISPATCH_LEVEL.
PO_FX_FLAG_ASYNC_ONLY 0x2	Make the condition change fully asynchronous. If this flag is set, the calling driver's callback routine is called from a thread other than the thread in which the routine that requests the condition change is called. Thus, the routine that requests the condition change always returns asynchronously without waiting for the callback to complete.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`PerfChange`

A pointer to a PO_FX_PERF_STATE_CHANGE structure that represents the new performance state of the component.

`Context`

A pointer to the context for the ComponentPerfStateCallback callback routine. This parameter is optional. It is provided so that a driver or device context can be passed to the callback routine. If this parameter is not used, it must be set to NULL.

Return Value

None

Remarks

When a driver calls **PoFxIssueComponentPerfStateChange**, the power management framework (PoFx) requests the platform extension plug-in (PEP) to place the component's performance state set in the specified performance state. This routine may be used with both discrete and range-based types of performance state sets. For more information about discrete and range-based performance state sets, see PO_FX_PERF_STATE_TYPE.

If *Flags* = **PO_FX_FLAG_BLOCKING**, the **PoFxIssueComponentPerfStateChange** call is synchronous. In this case, **PoFxIssueComponentPerfStateChange** waits to return until the component completes the performance state transition. The driver's ComponentPerfStateCallback routine is called to inform the driver that the component's performance state change is complete. This callback occurs in the same thread as the call to **PoFxIssueComponentPerfStateChange**, and **PoFxIssueComponentPerfStateChange** returns only after the *ComponentPerfStateCallback* callback returns.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY**, the **PoFxIssueComponentPerfStateChange** call is asynchronous. In this case, **PoFxIssueComponentPerfStateChange** schedules the ComponentPerfStateCallback routine to occur in another thread, and then returns without waiting for the callback to occur. The callback can occur before or after **PoFxIssueComponentPerfStateChange** returns. The driver should rely on the *ComponentPerfStateCallback* routine to determine when the component completes the transition to the new performance state.

The driver can set *Flags* = 0 to indicate that it does not care whether the **PoFxIssueComponentPerfStateChange** call is synchronous or asynchronous. In this case, PoFx decides the synchronicity of the call based on whether the PEP uses a synchronous or asynchronous request to issue the performance state change to the component.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY** or no flags are passed, this routine requires an IRQL of <= DISPATCH_LEVEL. If *Flags* = **PO_FX_FLAG_BLOCKING**, this routine requires an IRQL of <= APC_LEVEL.

This function will always result in a call to the ComponentPerfStateCallback routine regardless of the synchronicity of the call. Because the PEP may choose to deny the request to change the performance state, the driver must wait until receiving the callback before committing the performance state to hardware.

Only a single call of the **PoFxIssueComponentPerfStateChange** routine is allowed at a time per component, regardless of whether the call is synchronous or asynchronous. After issuing a performance state change request, the driver must wait until the ComponentPerfStateCallback is received before calling this routine again, even if the request involves a different performance state set. If this routine is called again before waiting until the *ComponentPerfStateCallback* is received, a bugcheck will occur.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= APC_LEVEL or <= DISPATCH_LEVEL (See Remarks section)"

See Also

ComponentPerfStateCallback

Device Performance State Management

PO_FX_PERF_STATE_TYPE

PoFxIssueComponentPerfStateChangeMultiple

PoFxRegisterComponentPerfStates

PoFxIssueComponentPerfStateChangeMultiple function

1/25/2019 • 3 minutes to read • Edit Online

The **PoFxIssueComponentPerfStateChangeMultiple** routine submits a request to change the performance states in multiple performance state sets simultaneously for a device component.

Syntax

```
void PoFxIssueComponentPerfStateChangeMultiple(
  POHANDLE                  Handle,
  ULONG                     Flags,
  ULONG                     Component,
  ULONG                     PerfChangesCount,
  PO_FX_PERF_STATE_CHANGE [] PerfChanges,
  PVOID                     Context
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`Flags`

The flags that modify the behavior of the performance state change operation. Set this member to zero or to one of the following flag **PO_FX_FLAG_*XXX*** bits:

These two flag bits are mutually exclusive. For more information, see Remarks.

VALUE	MEANING
PO_FX_FLAG_BLOCKING 0x1	Make the condition change synchronous. If this flag is set, the routine that requests the condition change does not return control to the calling driver until the component hardware completes the transition to the new condition. This flag can be used only if the caller is running at IRQL < DISPATCH_LEVEL.

PO_FX_FLAG_ASYNC_ONLY 0x2	Make the condition change fully asynchronous. If this flag is set, the calling driver's callback routine is called from a thread other than the thread in which the routine that requests the condition change is called. Thus, the routine that requests the condition change always returns asynchronously without waiting for the callback to complete.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`PerfChangesCount`

The number of performance state change requests contained in the *PerfChanges* array.

`PerfChanges`

The first element in an array of PO_FX_PERF_STATE_CHANGE structures that represent the performance states the driver intends to transition to. Each array element represents a single performance state change request.

`Context`

A pointer to the context for the ComponentPerfStateCallback callback routine. This parameter is optional. It is provided so that a driver or device context can be passed to the callback routine. If this parameter is not used, it must be set to NULL.

Return Value

None

Remarks

A driver calls **PoFxIssueComponentPerfStateChangeMultiple**, the power management framework (PoFx) will request the platform extension plug-in (PEP) to place the component's performance state sets in the specified performance states. This routine may be used with both discrete and range-based types of performance state sets. For more information about discrete and range-based performance state sets, see PO_FX_PERF_STATE_TYPE.

If *Flags* = **PO_FX_FLAG_BLOCKING**, the **PoFxIssueComponentPerfStateChangeMultiple** call is synchronous. In this case, **PoFxIssueComponentPerfStateChangeMultiple** waits to return until the component completes the performance state transition. The driver's ComponentPerfStateCallback routine is called to inform the driver that the component's performance state change is complete. This callback occurs in the same thread as the call to **PoFxIssueComponentPerfStateChangeMultiple**, and **PoFxIssueComponentPerfStateChangeMultiple** returns only after the *ComponentPerfStateCallback* callback returns.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY**, the **PoFxIssueComponentPerfStateChangeMultiple** call is asynchronous. In this case, **PoFxIssueComponentPerfStateChangeMultiple** schedules the ComponentPerfStateCallback routine to occur in another thread, and then returns without waiting for the callback to occur. The callback can occur before or after **PoFxIssueComponentPerfStateChangeMultiple** returns. The driver should rely on the *ComponentPerfStateCallback* routine to determine when the component completes the transition to the new performance state.

The driver can set *Flags* = 0 to indicate that it does not care whether the

PoFxIssueComponentPerfStateChangeMultiple call is synchronous or asynchronous. In this case, PoFx decides the synchronicity of the call based on whether the PEP uses a synchronous or asynchronous request to issue the performance state change to the component.

If *Flags* = **PO_FX_FLAG_ASYNC_ONLY** or no flags are passed, this routine requires an IRQL of <= DISPATCH_LEVEL. If *Flags* = **PO_FX_FLAG_BLOCKING**, this routine requires an IRQL of <= APC_LEVEL.

This function will always result in a call to the ComponentPerfStateCallback routine regardless of the synchronicity of the call. Because the PEP may choose to deny the request to change the performance states, the driver must wait until receiving the callback before committing the performance states to hardware.

Only a single call of the **PoFxIssueComponentPerfStateChangeMultiple** routine is allowed at a time per component, regardless of whether the call is synchronous or asynchronous. After issuing a performance state change request, the driver must wait until the ComponentPerfStateCallback is received before calling this routine again, even if the request involves a different performance state set. If this routine is called again before waiting until the *ComponentPerfStateCallback* is received, a bugcheck will occur.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= APC_LEVEL or <= DISPATCH_LEVEL (See Remarks section)"

See Also

ComponentPerfStateCallback

Device Performance State Management

PO_FX_PERF_STATE_TYPE

PoFxIssueComponentPerfStateChange

PoFxRegisterComponentPerfStates

PoFxNotifySurprisePowerOn function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxNotifySurprisePowerOn** routine notifies the power management framework (PoFx) that a device was turned on as a side effect of supplying power to some other device.

Syntax

```
void PoFxNotifySurprisePowerOn(
  PDEVICE_OBJECT Pdo
);
```

Parameters

`Pdo`

A pointer to a physical device object (PDO). This parameter points to a DEVICE_OBJECT structure that represents the physical device that was turned on as a side effect. The caller is always the bus driver that enumerated the PDO.

Return Value

None.

Remarks

Device drivers should not call this routine. This routine should be called only by bus drivers.

A bus driver calls this routine to inform PoFx that a device that is not currently being used was incidentally turned on at the same time as a second device. For example, the first device might share a power rail with the second device. Thus, power cannot be supplied to the second device without supplying power, as a side effect, to the first device. Because the first device is not being used, this device should be configured to consume as little power as possible.

To request a transition to a D0 power state, a device driver sends an IRP_MN_SET_POWER request down its device stack. Through the PDO in this stack, the bus driver that is the parent of this device receives the request and responds by supplying power to the device. However, if the bus driver cannot turn on this device without also turning on a second, unused device, the bus driver can call **PoFxNotifySurprisePowerOn** to reduce the power consumed by the unused device.

On entry to **PoFxNotifySurprisePowerOn**, the device represented by the *Pdo* parameter is in an uninitialized D0 power state. In this state, all of the components in the device are typically turned on. In response to the **PoFxNotifySurprisePowerOn** call, PoFx configures the device in an initialized D0 state. During this configuration, PoFx switches as many components as it can to low-power Fx power states. If possible, PoFx configures the device in a "hot D3" state, which is really a D0 state in which all of the individual components in the device are turned off.

PoFxNotifySurprisePowerOn can configure only a device that was registered with PoFx when the device was previously turned off. However, unless the bus driver knows that a device is not registered, the bus driver should call **PoFxNotifySurprisePowerOn** when the device is turned on as a side effect.

If the bus driver fails to call this routine when the device is turned on, the device hardware might stay in the fully on state for an indefinite time, during which PoFx assumes that the device remains in the D3 (fully off) power state.

Call **PoFxNotifySurprisePowerOn** only if the device was turned on incidentally, as a side effect of turning on some other device. If the bus driver restores power to a device in response to a DevicePowerRequiredCallback callback or an IRP_MN_SET_POWER request for a D0 transition, call the PoFxReportDevicePoweredOn routine instead to inform PoFx when power is restored to the device.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DEVICE_OBJECT

DevicePowerRequiredCallback

IRP_MN_SET_POWER

PoFxReportDevicePoweredOn

PoFxPowerControl function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxPowerControl** routine sends a power control request to the power management framework (PoFx).

Syntax

```
NTSTATUS PoFxPowerControl(
  POHANDLE Handle,
  LPCGUID  PowerControlCode,
  PVOID    InBuffer,
  SIZE_T   InBufferSize,
  PVOID    OutBuffer,
  SIZE_T   OutBufferSize,
  PSIZE_T  BytesReturned
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`PowerControlCode`

A pointer to the power control code. This code is a GUID value that specifies the requested operation.

`InBuffer`

A pointer to a caller-allocated buffer that contains the input data for the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *InBuffer* parameter is optional and can be specified as NULL if the specified operation requires no input data.

`InBufferSize`

The size, in bytes, of the input buffer that is pointed to by the *InBuffer* parameter. If *InBuffer* is NULL, set *InBufferSize* to zero.

`OutBuffer`

A pointer to a caller-allocated buffer that is to contain the output data from the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *OutBuffer* parameter is optional and can be specified as NULL if the specified operation produces no output data.

`OutBufferSize`

The size, in bytes, of the output buffer that is pointed to by the *OutBuffer* parameter. If *OutBuffer* is NULL, set *OutBufferSize* to zero.

`BytesReturned`

A pointer to a location into which the routine writes the number of bytes of data that were written to the buffer that is pointed to by *OutBuffer*. The number of bytes written will be less than or equal to *OutBufferSize*. This parameter is optional and can be specified as **NULL** if the caller does not need to know how many bytes were written to the

output buffer.

Return Value

PoFxPowerControl returns **STATUS_SUCCESS** if the requested operation succeeds. Possible error return values include the following status code.

RETURN CODE	DESCRIPTION
STATUS_NOT_IMPLEMENTED	The requested power control operation is not implemented.
STATUS_NOT_SUPPORTED	The power engine plug-in (PEP) does not acknowledge support for this device.

Remarks

A device driver calls this routine to send a power control request directly to PoFx. A power control request is similar to an I/O control request (IOCTL). Unlike an IOCTL, however, a power control request is sent directly to PoFx and is not observed by other device drivers in the device stack. During a **PoFxPowerControl** call, PoFx synchronously performs the requested operation.

Similarly, PoFx can send a power control request directly to the device driver. The driver handles this request in its PowerControlCallback routine.

PoFx delegates the handling of all power control requests to the power engine plug-in (PEP). The PEP is an optional software component that performs power management tasks that are specific to a particular product line of processor or System on a Chip (SoC) modules. If the hardware vendor for the processor or SoC supplies a PEP for a hardware platform, this PEP might handle custom power control requests from a device driver, or might send custom power control requests to the driver's *PowerControlCallback* routine. The vendor can specify a set of *PowerControlCode* GUIDs and define the operations that are designated by these GUIDs. As an option, a device driver can contain platform-specific code to handle or to send requests for these operations.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h

Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

PoFxRegisterDevice

PowerControlCallback

PoFxPowerOnCrashdumpDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxPowerOnCrashdumpDevice** routine requests that a crash-dump device be turned on.

Syntax

```
NTSTATUS PoFxPowerOnCrashdumpDevice(
  POHANDLE Handle,
  PVOID    Context
);
```

Parameters

`Handle`

A handle that represents the registration of the crash-dump device with the Windows power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Context`

A pointer to a device-specific context. This pointer is passed as an input parameter to the PowerOnDumpDeviceCallback callback routine that is implemented by the platform extension plug-in (PEP) for the device. The context information is stored in a format that is defined by the device driver and is understood by the PEP. This context is opaque to the operating system. The driver can set this parameter to NULL if the PEP does not require a context.

Return Value

PoFxPowerOnCrashdumpDevice returns STATUS_SUCCESS if the routine succeeds in turning on power to the device. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	Not a valid handle value.
STA TUS _UN SUC CES SFU L	The PEP for this device does not implement a *PowerOnDumpDeviceCallback* callback routine; or the PEP failed to turn on the device.

Remarks

The driver for a crash-dump device calls this routine to request that the platform extension plug-in (PEP) turn the device on so that a crash dump can be saved. All devices in the crash-dump device chain (which might include a storage controller, a PCI controller, and so on) must be turned on before a crash-dump file can be written to disk.

For more information about crash dumps, see Kernel-Mode Dump Files.

This routine can be called at IRQL <= HIGH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Windows
Header	wdm.h (include Pepfx.h)
Library	Ntoskrnl.lib
IRQL	"<= HIGH_LEVEL"

See Also

PoFxRegisterDevice

PowerOnDumpDeviceCallback

PoFxQueryCurrentComponentPerfState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxQueryCurrentComponentPerfState** routine retrieves the active performance state in a component's performance state set.

Syntax

```
NTSTATUS PoFxQueryCurrentComponentPerfState(
  POHANDLE   Handle,
  ULONG      Flags,
  ULONG      Component,
  ULONG      SetIndex,
  PULONGLONG CurrentPerf
);
```

Parameters

`Handle`

A handle that represents the registration of the device with power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Flags`

Set to 0. Currently, no flags are defined for this routine.

`Component`

The index that identifies the component whose performance state is being retrieved. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`SetIndex`

The index that identifies the performance state set whose performance is being queried. This parameter is an index into the **PerfStateSets** array in the PO_FX_COMPONENT_PERF_INFO structure that the device driver used to register the component performance states with PoFx. If the **PerfStateSets** array contains N elements, performance state set indexes range from 0 to N−1.

`CurrentPerf`

A pointer to variable in which the current performance state for the specified performance state set is returned. For more information about the value that is returned, see Remarks.

Return Value

PoFxQueryCurrentComponentPerfState returns **STATUS_SUCCESS** if the requested operation succeeds. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION

STA TUS _IN VAL ID_ PAR AM ETE R	The caller supplied an invalid index for the component or component state set.

Remarks

A driver can call the **PoFxQueryCurrentComponentPerfState** routine to retrieve the current performance state of a given performance state set for a component. If the call succeeds, the *CurrentPerf* parameter is overwritten with a pointer to a value that specifies the current performance state of the set. This value may be one of the following, depending on the contents of the PO_FX_COMPONENT_PERF_SET structure that the driver passed to the PoFxRegisterComponentPerfStates routine:

- For a performance state set that represents a continuous distribution of performance states, this is a value between the **Minimum** and **Maximum** members of the PO_FX_COMPONENT_PERF_SET structure.
- For a performance state set that represents a discrete number of performance states, this value is an index into the **States** member of the PO_FX_COMPONENT_PERF_SET structure .

This call always executes synchronously.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

Device Performance State Management

PO_FX_COMPONENT_PERF_INFO

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

PoFxRegisterComponentPerfStates function

1/25/2019 • 4 minutes to read • Edit Online

The **PoFxRegisterComponentPerfStates** routine registers a device component for performance state management by the power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterComponentPerfStates(
  POHANDLE                              Handle,
  ULONG                                 Component,
  ULONGLONG                             Flags,
  PPO_FX_COMPONENT_PERF_STATE_CALLBACK ComponentPerfStateCallback,
  PPO_FX_COMPONENT_PERF_INFO           InputStateInfo,
  PPO_FX_COMPONENT_PERF_INFO           *OutputStateInfo
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component whose performance states will be managed. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N−1.

`Flags`

The flags that modify the behavior of the performance state registration. Set this member to zero or to one of the following flag **PO_FX_FLAG_PERF_*XXX*** bits:

VALUE	MEANING
PO_FX_FLAG_PERF_POTION_AL 0x1	Indicates that the driver can change performance states without assistance from the platform extension plug-in (PEP), or that the driver is registering performance states with PoFx for logging purposes only. If this flag is set, the **PoFxRegisterComponentPerfStates** call will still succeeded if the PEP does not support performance states for the component.

PO_FX_FLAG_PERF_QUERY_ON_F0 0x2	For some devices, the PEP may need to a place a performance state set for a component into a certain performance state (known as a *nominal performance state*) when it idles the component. Drivers set this flag if the component contains nominal performance states, in which case PoFx will query the PEP to determine the current performance state when the component transitions to F0.
PO_FX_FLAG_PERF_QUERY_ON_ALLID LE_STATES 0x4	For some devices, the PEP may need to a place a performance state set for a component into a certain performance state (known as a *nominal performance state*) when it transitions the component between idle states. Drivers set this flag if this component contains nominal performance states, in which case PoFx will query the PEP to determine the current performance state when the component transitions between idle states.

`ComponentPerfStateCallback`

A pointer to a ComponentPerfStateCallback routine. This routine is called when PoFx has completed logging and notifying the PEP about a performance state transition that is initiated by the driver's call to PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple. This callback may be the same for all components and all devices; PoFx provides the device handle and component index in each completion call.

`InputStateInfo`

If the driver provides performance state info, this parameter contains a pointer to a driver allocated PO_FX_COMPONENT_PERF_INFO structure that provides performance state information to PoFx. If the driver requires performance state information from the PEP, this parameter must be set to NULL.

`OutputStateInfo`

If the driver requires performance state information from the PEP, after a successful registration this parameter contains a pointer to a PO_FX_COMPONENT_PERF_INFO structure that provides performance state information defined by the PEP. If the driver provides performance state info, this parameter must be set to NULL.

The memory allocated for this parameter is managed by PoFx, and the driver should not free this memory when the device is removed. The lifetime of this memory is guaranteed to exceed the lifetime of the PoFx component that contains these performance state sets.

Return Value

PoFxRegisterComponentPerfStates returns **STATUS_SUCCESS** if PoFx accepts the device's registration of performance states. If any of the necessary information is not provided or incorrect, registration will fail with a return code other than **STATUS_SUCCESS**. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_NOT_IMPLEMENTED	The *Flags* parameter does not include the **PO_FX_FLAG_PERF_PEP_OPTIONAL** flag and the PEP is not able to provide performance state management for this device.
STATUS_INVALID_PARAMETER	Both *InputStateInfo* and *OutputStateInfo* are NULL or both of these parameters are not NULL, or there are no performance state sets in the PO_FX_COMPONENT_PERF_INFO structure that was assigned to the *InputStateInfo* parameter.

Remarks

Note that PoFx performs a deep-copy of the device's structures into its memory.

Either the driver or the platform extension plug-in (PEP) may provide information about the performance states supported by each component:

- If the driver provides performance state information, the driver must set the *InputStateInfo* parameter to a pointer to a PO_FX_COMPONENT_PERF_INFO structure that contains the performance state information. Otherwise, the driver must set this parameter to NULL.
- If the PEP provides performance state information, the driver must set the *OutputStateInfo* parameter to a valid pointer to a PO_FX_COMPONENT_PERF_INFO structure that receives the performance state information. Otherwise, the driver must set this parameter to NULL.

If the PEP does not support performance states, the driver may register for performance state support with PoFx for logging purposes only.

If the driver registers for performance state support for logging purposes only, or if the driver can function correctly with or without PEP support for performance state management, the driver must set the **PO_FX_FLAG_PERF_PEP_OPTIONAL** flag in the *Flags* parameter. If the flag is set, the registration call will succeed even if the PEP does not provide support for performance states.

If the driver requires the PEP to provide performance state information, the driver cannot set the **PO_FX_FLAG_PERF_PEP_OPTIONAL** flag in the *Flags* parameter.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal

Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= APC_LEVEL"

See Also

ComponentPerfStateCallback

Device Performance State Management

PO_FX_COMPONENT_PERF_INFO

PoFxRegisterDevice

PoFxRegisterCrashdumpDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxRegisterCrashdumpDevice** routine registers a crash-dump device.

Syntax

```
NTSTATUS PoFxRegisterCrashdumpDevice(
  POHANDLE Handle
);
```

Parameters

`Handle`

A handle that represents the registration of the crash-dump device with the Windows power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

Return Value

PoFxRegisterCrashdumpDevice returns STATUS_SUCCESS if the routine successfully registers the crash-dump device. Possible error return values include the following status codes.

RETURN VALUE	DESCRIPTION
STA TUS _IN VAL ID_P ARA MET ER	Not a valid handle value.
STA TUS _UN SUC CES SFU L	There is no PEP for this device.

Remarks

This routine is called by the driver for a crash-dump device to inform PoFx that the device is part of the crash-dump device chain. Several devices (storage controller, PCI controller, and so on) might need to be turned on so that the Windows kernel can write a crash-dump file to disk. When a fatal system error occurs, the kernel tries to turn on the crash-dump devices and save the crash-dump file.

The driver must call PoFxRegisterDevice to register the device with PoFx before calling **PoFxRegisterCrashdumpDevice**.

The driver for a crash-dump device can call the PoFxPowerOnCrashdumpDevice routine to request that the PEP turn the device on.

For more information about crash dumps, see Kernel-Mode Dump Files.

PoFxRegisterCrashdumpDevice must be called at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 8.1.
Target Platform	Windows
Header	wdm.h (include Pepfx.h)
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

PoFxPowerOnCrashdumpDevice

PoFxRegisterDevice

PoFxRegisterDevice function

1/25/2019 • 3 minutes to read • Edit Online

The **PoFxRegisterDevice** routine registers a device with the power management framework (PoFx).

Syntax

```
NTSTATUS PoFxRegisterDevice(
  PDEVICE_OBJECT Pdo,
  PPO_FX_DEVICE  Device,
  POHANDLE       *Handle
);
```

Parameters

`Pdo`

A pointer to a physical device object (PDO). This parameter points to a DEVICE_OBJECT structure that represents the physical device that is being registered. The caller is the power policy owner for the device, which is typically the device's function driver.

`Device`

A pointer to a caller-allocated PO_FX_DEVICE structure that contains the registration information for the device. This structure contains pointers to a set of callback routines that are implemented by the device driver. PoFx calls these routines to communicate with the driver.

`Handle`

A pointer to a location into which the routine writes a handle that represents the registration of the device with PoFx. The device driver passes this handle as an input parameter to the other **PoFx*Xxx*** routines that it calls. The driver must first call **PoFxRegisterDevice** to register the device before the driver calls any other **PoFx*Xxx*** routines to power-manage the device.

Return Value

PoFxRegisterDevice returns **STATUS_SUCCESS** if the routine successfully registers the device. Possible error return values include the following status codes.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	*Pdo* is NULL; or the **PPO_FX_DEVICE** structure has an invalid version number or a component count of zero; or the number of idle states specified for a component is zero; or the description of an idle state is invalid.

STATUS_DEVICE_NOT_READY	The device is not ready.
STATUS_INSUFFICIENT_RESOURCES	Insufficient resources are available to complete the registration.

Remarks

A device driver typically calls this routine from the driver's IRP_MN_START_DEVICE request handler. The driver must not call this routine before the device receives an **IRP_MN_START_DEVICE** request. The device receives the first **IRP_MN_START_DEVICE** request when the device is being started for the first time. The device receives an additional **IRP_MN_START_DEVICE** request each time the device is restarted after being stopped for resource balancing. A **PoFxRegisterDevice** call to register a device that is already registered is a fatal error and causes a bug check. Note that PoFx performs a deep-copy of the device structures into its memory.

Before the driver calls **PoFxRegisterDevice**, the device must meet the following conditions:

- The device (that is, the PDO) is not already registered with PoFx.
- The device is in the D0 (fully on) power state.
- The device is in the running condition.
- Every component in the device is in the F0 (fully on) power state.
- Every component is in the active condition.

By registering the device with PoFx, the driver assumes the responsibility for informing PoFx when a component is actively being used and when the component is idle. While the device is registered, the driver must call the PoFxActivateComponent routine to gain access to a component's hardware registers, and the driver must call the PoFxIdleComponent routine to notify PoFx when the driver no longer requires access to the component.

After a driver calls **PoFxRegisterDevice** to register a device with PoFx, all components in the device are fully on and in the active condition so that the driver can finish initializing the hardware. To start active power management, the driver must call the PoFxStartDevicePowerManagement routine.

By default, **PoFxStartDevicePowerManagement** switches all components to the idle condition. If the driver requires a component to be in the active condition immediately after power management starts, the driver must explicitly activate the component by calling the **PoFxActivateComponent** routine, and this call must occur after the **PoFxRegisterDevice** call but before the **PoFxStartDevicePowerManagement** call.

Typically, the Kernel-Mode Driver Framework (KMDF) driver for a single-component device does not call **PoFxRegisterDevice** to register the device with PoFx. Instead, this driver receives a PoFx registration handle when KMDF calls the driver's EvtDeviceWdmPostPoFxRegisterDevice callback function. For more information, see

Supporting Multiple Functional Power States for Single-Component Devices.

For information about how the KMDF driver for a multiple-component device registers with PoFx, see Supporting Multiple Functional Power States for Multiple-Component Devices.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	PASSIVE_LEVEL

See Also

DEVICE_OBJECT

EvtDeviceWdmPostPoFxRegisterDevice

IRP_MN_START_DEVICE

PO_FX_DEVICE

PoFxActivateComponent

PoFxIdleComponent

PoFxStartDevicePowerManagement

PoFxReportDevicePoweredOn function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxReportDevicePoweredOn** routine notifies the power management framework (PoFx) that the device completed the requested transition to the D0 (fully on) power state.

Syntax

```
void PoFxReportDevicePoweredOn(
  POHANDLE Handle
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

Return Value

None.

Remarks

The driver for a registered device must call **PoFxReportDevicePoweredOn** after either of the following occurrences:

- PoFx calls the driver's DevicePowerRequiredCallback callback routine to notify the driver that the device must enter the D0 power state.
- PoFx sends the driver an S0 IRP (an IRP_MN_SET_POWER request) to notify the driver of a pending system transition to the S0 power state.

In response to either occurrence, the driver sends D0 IRP (an **IRP_MN_SET_POWER** request) down its device stack to initiate a transition to the D0 power state, if the device is not already in the D0 state. After the driver completes (and all lower drivers complete) the transition to the D0 state, the driver calls **PoFxReportDevicePoweredOn** to notify PoFx.

Each time the *DevicePowerRequiredCallback* routine is called, the driver must respond by calling **PoFxReportDevicePoweredOn**. The **PoFxReportDevicePoweredOn** call can occur either during or after the *DevicePowerRequiredCallback* call. The driver must call **PoFxReportDevicePoweredOn** regardless of whether no D0 IRP is required, the D0 IRP request fails, or the dispatched D0 IRP succeeds or fails.

During a system transition to the S0 state, PoFx sends the driver an **IRP_MN_SET_POWER** request to notify the driver of the pending S0 transition. In response, the driver must complete this request as soon as possible, switch the device to the D0 power state, and call **PoFxReportDevicePoweredOn**. For more information, see Handling a System Set-Power IRP in a Device Power Policy Owner.

On entry to **PoFxReportDevicePoweredOn**, the device might be in an uninitialized D0 power state in which all of the components in the device are turned on. In response to the **PoFxReportDevicePoweredOn** call, PoFx configures the device in an initialized D0 state. During this configuration, PoFx switches as many components as it

can to low-power Fx power states.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DevicePowerRequiredCallback

IRP_MN_SET_POWER

PoFxRegisterDevice

PoFxSetComponentLatency function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxSetComponentLatency** routine specifies the maximum latency that can be tolerated in the transition from the idle condition to the active condition in the specified component.

Syntax

```
void PoFxSetComponentLatency(
  POHANDLE  Handle,
  ULONG     Component,
  ULONGLONG Latency
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Latency`

The maximum latency, in 100-nanosecond units, that can be tolerated in the specified component. For more information, see Remarks.

Return Value

None.

Remarks

The calling driver supplies a maximum latency that clients of the device can tolerate in the specified component. This latency is the time required to complete a transition from the idle condition to the active condition. PoFx uses this latency as a hint to improve performance.

When the component switches from the active condition to the idle condition, PoFx selects an appropriate low-power Fx state for the component to enter. After this transition completes, the time required to switch back to the active condition depends on the selected Fx state. PoFx tries to select an Fx state that does not violate the maximum latency requirement.

For example, if a driver supports power states F0, F1, and F2 in the specified component, the F2 state represents a lower power setting than the F1 state. Typically, the time T2 required to wake (to the fully on state, F0) from F2 is longer than the time T1 required to wake from F1. If the maximum latency is greater than T1 but less than T2, PoFx can safely switch the component to the F1 state and still meet the maximum latency requirement. However, PoFx avoids switching the component to the F2 state because doing so violates this requirement.

The device driver can call **PoFxSetComponentLatency** each time a change in conditions requires a change in the maximum latency. After each call, the new maximum latency remains in effect until the driver calls **PoFxSetComponentLatency** again to update it.

If a component is in the idle condition when **PoFxSetComponentLatency** is called, PoFx might change the component's current Fx state to accommodate the new maximum latency specified by the caller.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

PO_FX_DEVICE

PoFxRegisterDevice

PoFxSetComponentResidency function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxSetComponentResidency** routine sets the estimated time for how long a component is likely to remain idle after the component enters the idle condition.

Syntax

```
void PoFxSetComponentResidency(
  POHANDLE  Handle,
  ULONG     Component,
  ULONGLONG Residency
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Residency`

The estimated residency time, in 100-nanosecond units. This parameter is a hint to PoFx about how long the component is likely to remain idle after a transition from the active condition to the idle condition. For more information, see Remarks.

Return Value

None.

Remarks

The calling driver supplies an estimated residency time that PoFx can use as a hint to improve performance. PoFx uses this hint to select an appropriate low-power Fx state for a component that is in the idle condition.

The device driver can call **PoFxSetComponentResidency** each time a change in circumstances requires a change in the estimated residency time. After each call, the new estimated residency time remains in effect until the driver calls **PoFxSetComponentResidency** again to update it.

If a component is in the idle condition when **PoFxSetComponentResidency** is called, PoFx might change the component's Fx state to accommodate the new estimated residency time specified by the caller.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

PoFxSetComponentWake function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxSetComponentWake** routine indicates whether the driver arms the specified component to wake whenever the component enters the idle condition.

Syntax

```
void PoFxSetComponentWake(
  POHANDLE Handle,
  ULONG    Component,
  BOOLEAN  WakeHint
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the PO_FX_DEVICE structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`WakeHint`

Whether the driver arms the component to wake whenever the component enters the idle condition. If TRUE, the driver arms the component to wake. If FALSE, the driver does not arm the component to wake.

Return Value

None.

Remarks

Before a component enters a low-power Fx state, the driver might arm the component to automatically wake in response to an input signal from an external source (for example, if the user pushes a button). The *WakeHint* parameter informs PoFx whether the driver arms the component to wake. If a component is armed to wake, PoFx will not switch the component to a low-power Fx state from which the component cannot wake.

When a device is registered with PoFx, the registration information for each component in the device is contained in a separate PO_FX_COMPONENT structure. The **DeepestWakeableIdleState** member of this structure specifies the deepest Fx state from which the component can wake.

The device driver can call **PoFxSetComponentWake** each time a change in conditions requires a change in the *WakeHint* value. After each call, the new *WakeHint* value remains in effect until the driver calls **PoFxSetComponentWake** again to update it.

If a component is in the idle condition when **PoFxSetComponentWake** is called, PoFx might change the

component's Fx state to accommodate the new *WakeHint* parameter value specified by the caller.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

PO_FX_COMPONENT

PoFxSetDeviceIdleTimeout function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxSetDeviceIdleTimeout** routine specifies the minimum time interval from when the last component of the device enters the idle condition to when the power management framework (PoFx) calls the driver's DevicePowerNotRequiredCallback routine.

Syntax

```
void PoFxSetDeviceIdleTimeout(
  POHANDLE  Handle,
  ULONGLONG IdleTimeout
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`IdleTimeout`

The idle time-out interval in 100-nanosecond units. For more information, see Remarks.

Return Value

None.

Remarks

This routine specifies a time-out interval for PoFx to apply to future calls to the driver's DevicePowerNotRequiredCallback callback routine. By default, this time-out interval is zero, in which case PoFx might call the *DevicePowerNotRequiredCallback* routine just as soon as the device is ready to switch to a low-power Dx state. However, a driver might prefer to delay this transition and to keep the device in the D0 power state for some additional time-out interval. In this case, if the device becomes active before the end of the time-out interval, and is therefore required to stay in the D0 state, the pending *DevicePowerNotRequiredCallback* call is no longer required and is canceled by PoFx.

For example, to improve performance and reduce wear, a storage device driver might want to prevent a spinning drive from entering D3 and slowing down until the drive has been idle for a sufficiently long time. To simplify this driver's implementation of the *DevicePowerNotRequiredCallback* callback routine, the **PoFxSetDeviceIdleTimeout** routine enables the driver to automatically delay the D3 transition without requiring the driver to set up a timer.

The time-out interval starts when all components of the device complete their transitions to the idle condition. Typically, PoFx waits until the end of the time-out interval to call the *DevicePowerNotRequiredCallback* routine. However, if PoFx is preparing to enter a low-power system state, PoFx might end the time-out interval early. In any case, when PoFx calls the driver's *DevicePowerNotRequiredCallback* routine, the driver should switch to the low-power Dx state without further delay.

The device driver can call **PoFxSetDeviceIdleTimeout** each time a change in conditions requires a change in the idle time-out interval. After each call, the new idle time-out interval remains in effect until the driver calls **PoFxSetDeviceIdleTimeout** again to update it.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DevicePowerNotRequiredCallback

PoFxRegisterDevice

PoFxSetTargetDripsDevicePowerState function

1/25/2019 • 2 minutes to read • Edit Online

This routine is called to notify the power manager of the device's target device power state for DRIPS. The driver can override the DRIPS constraint provided by the PEP.

Syntax

```
NTSTATUS PoFxSetTargetDripsDevicePowerState(
  POHANDLE           Handle,
  DEVICE_POWER_STATE TargetState
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

`TargetState`

Specifies the target DRIPS device power state. Possible values are defined in the DEVICE_POWER_STATE enumeration. This value must be lower than the existing device constraint. A device power state of **PowerDeviceUnspecified** resets to the PEP provided constraint.

Return Value

Returns STATUS_SUCCESS if the target state was accepted.

Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode)
IRQL	PASSIVE_LEVEL

PoFxStartDevicePowerManagement function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxStartDevicePowerManagement** routine completes the registration of a device with the power management framework (PoFx) and starts device power management.

Syntax

```
void PoFxStartDevicePowerManagement(
  POHANDLE Handle
);
```

Parameters

`Handle`

A handle that represents the registration of the device with the power management framework (PoFx). The device driver previously received this handle from the PoFxRegisterDevice routine.

Return Value

None.

Remarks

After a driver calls the PoFxRegisterDevice routine to register a device with PoFx, all components in the device are fully on and in the active condition so that the driver can finish initializing the hardware. To start active power management, the driver must call **PoFxStartDevicePowerManagement**. By default, this call switches all components to the idle condition.

After **PoFxStartDevicePowerManagement** switches a component to the idle condition, PoFx might immediately start calling the driver's callback routines, even before **PoFxStartDevicePowerManagement** returns. PoFx calls the driver's ComponentIdleConditionCallback callback routine to indicate that a component has switched to the idle condition. The driver must not access a component that is in the idle condition. Before the driver can access the component, the driver must call the PoFxActivateComponent and wait for PoFx to call the driver's ComponentActiveConditionCallback callback routine. This callback informs the driver that the component is in the active condition and can safely be accessed.

If the driver requires a component to be in the active condition immediately after power management starts, the driver must explicitly activate the component by calling the **PoFxActivateComponent** routine, and this call must occur after the **PoFxRegisterDevice** call but before the **PoFxStartDevicePowerManagement** call.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal

Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ComponentActiveConditionCallback

ComponentIdleConditionCallback

PoFxActivateComponent

PoFxRegisterDevice

PoFxUnregisterDevice function

1/25/2019 • 2 minutes to read • Edit Online

The **PoFxUnregisterDevice** routine removes the registration of a device from the power management framework (PoFx).

Syntax

```
void PoFxUnregisterDevice(
  POHANDLE Handle
);
```

Parameters

`Handle`

A handle that represents the registration of the device with PoFx. The device driver previously received this handle from the PoFxRegisterDevice routine.

Return Value

None.

Remarks

The driver that owns the power policy for a device must unregister the device from PoFx when the device is removed. The Plug and Play (PnP) manager can remove the device stack from the PnP tree only after the device is unregistered.

To unregister the device, the driver calls **PoFxUnregisterDevice** during the handling of an IRP_MN_REMOVE_DEVICE or IRP_MN_SURPRISE_REMOVAL request, whichever is received first. This call should occur after the driver no longer requires access to the device's hardware.

A device driver that calls this routine must previously have called the PoFxRegisterDevice routine to register the device with PoFx.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe

IRQL	PASSIVE_LEVEL

See Also

IRP_MN_REMOVE_DEVICE

IRP_MN_SURPRISE_REMOVAL

PoFxRegisterDevice

PoGetSystemWake function

1/25/2019 • 2 minutes to read • Edit Online

The **PoGetSystemWake** routine determines whether a specified IRP has been marked as waking the system from a sleeping state.

Syntax

```
BOOLEAN PoGetSystemWake(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to an IRP.

Return Value

PoGetSystemWake returns one of the following:

RETURN CODE	DESCRIPTION
TRUE	The specified IRP did cause the system to wake.
FALSE	The specified IRP did not cause the system to wake.

Remarks

A driver calls **PoGetSystemWake** to determine if a specified IRP contributed to waking the system from a sleep state.

A driver in a wait/wake chain should call **PoGetSystemWake** on its own wait/wake IRP at completion to determine if the driver should also call PoSetSystemWake for child wait/wake IRPs that the driver is about to complete. This ensures that system wake information properly propagates throughout the entire wait/wake chain.

It is possible that several IRPs are causing the system to wake. In this case, **PoGetSystemWake** would return **TRUE** for all of the IRPs contributing to the wake event.

Requirements

Minimum supported client	Available starting with Windows Vista.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

PoSetSystemWake

PopEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **PopEntryList** routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
PSINGLE_LIST_ENTRY PopEntryList(
  PSINGLE_LIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the SINGLE_LIST_ENTRY structure that represents the head of the list. On return, *ListHead*->**Next** points to the beginning of the list with the first entry removed.

Return Value

PopEntryList returns a pointer to the entry removed from the list, or **NULL** if the list is currently empty.

Remarks

PopEntryList removes the first entry from the list by setting *ListHead*->**Next** to point to the second entry in the list.

For information about using this routine when implementing a singly linked list, see Singly and Doubly Linked Lists.

Callers of **PopEntryList** can be running at any IRQL. If **PopEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedPopEntryList

PushEntryList

PoQueryWatchdogTime function

1/25/2019 • 2 minutes to read • Edit Online

The **PoQueryWatchdogTime** routine indicates whether the power manager has enabled a watchdog time-out counter for any power IRP that is currently assigned to the device stack.

Syntax

```
BOOLEAN PoQueryWatchdogTime(
  PDEVICE_OBJECT Pdo,
  PULONG         SecondsRemaining
);
```

Parameters

`Pdo`

A pointer to a physical device object (PDO). This parameter points to a DEVICE_OBJECT structure that represents a physical device.

`SecondsRemaining`

A pointer to a location into which the routine writes the time, in seconds, that remains before the next power watchdog time-out is set to occur.

Return Value

PoQueryWatchdogTime returns **TRUE** if a watchdog-enabled power IRP is currently assigned to the device stack. Otherwise, it returns **FALSE**.

Remarks

This routine enables kernel-mode drivers to monitor watchdog time-out counters that the power manager has enabled to keep track of power IRPs that it has issued. If one or more watchdog time-out counters are currently enabled, the routine returns **TRUE** and provides the amount of time that remains before the next time-out.

For example, a driver that experiences delays when shutting down a device can call this routine to determine how much time remains before the driver must respond to a power IRP to prevent a controlled shutdown (a bug check) of the operating system.

The power manager sets a watchdog time-out counter when it issues a power IRP to the device stack. The time-out period for this counter is typically several minutes. If a device in the stack is unresponsive and causes the IRP to stall for the time-out period, the power manager treats this condition as an unrecoverable error and initiates a controlled shutdown of the operating system.

If more than one power watchdog time-out is currently enabled, the routine sets **SecondsRemaining* to the time that remains to the next time-out.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

DEVICE_OBJECT

PoRegisterDeviceForIdleDetection function

1/25/2019 • 3 minutes to read • Edit Online

The **PoRegisterDeviceForIdleDetection** routine enables or cancels idle detection and sets idle time-out values for a device.

Syntax

```
PULONG PoRegisterDeviceForIdleDetection(
  PDEVICE_OBJECT     DeviceObject,
  ULONG              ConservationIdleTime,
  ULONG              PerformanceIdleTime,
  DEVICE_POWER_STATE State
);
```

Parameters

`DeviceObject`

A pointer to the driver-created DEVICE_OBJECT for the device. On Windows 2000 and later systems, this parameter can point to a physical device object (PDO) or a functional device object (FDO). On Windows 98/Me, this parameter must point to the PDO of the underlying device.

`ConservationIdleTime`

Sets the time-out value (in seconds) to apply when the system power policy optimizes for energy conservation. Specify zero to disable idle detection when conservation policy is in effect.

`PerformanceIdleTime`

Sets the time-out value (in seconds) to apply when the system power policy optimizes for performance. Specify zero to disable idle detection when performance policy is in effect.

`State`

Specifies the device power state to be requested in an IRP_MN_SET_POWER request when either *ConservationIdleTime* or *PerformanceIdleTime* has been met. Possible values are the DEVICE_POWER_STATE values.

Return Value

PoRegisterDeviceForIdleDetection returns a pointer to the idle counter to indicate that idle detection has been enabled. It returns **NULL** to indicate that idle detection has been disabled, that an idle counter could not be allocated, or that one or both of the time-out values were invalid.

Remarks

PoRegisterDeviceForIdleDetection enables drivers to use the idle detection mechanism provided by the power manager. Drivers call **PoRegisterDeviceForIdleDetection** for any of the following reasons:

- To enable idle detection for the device and set initial idle time-out values
- To change the idle time-out values for a device

- To disable idle detection for a device

After enabling a device for idle detection, a driver calls PoSetDeviceBusy whenever the device is in use, passing the non-**NULL** idle pointer returned by **PoRegisterDeviceForIdleDetection**. Calling **PoSetDeviceBusy** restarts the idle countdown. Note that a driver must not pass a **NULL** pointer to **PoSetDeviceBusy**.

Whenever the device satisfies the current idle time-out value, the power manager sends an **IRP_MN_SET_POWER** request to the top of the device stack, specifying device power state *State*. In response to the IRP, each driver performs any device-specific tasks required before the power state transition, then passes the IRP to the next-lower driver. When the IRP reaches the bus driver, that driver puts the device in the requested lower power state and completes the IRP.

PoRegisterDeviceForIdleDetection sets time-out values for both conservation and performance. The *ConservationIdleTime* value applies when the system power policy optimizes for conservation; the *PerformanceIdleTime* value applies when the system power policy optimizes for performance. Typically, the applicable policy depends upon the power source: when running with AC power, the system optimizes for performance, and when running off a battery, the system optimizes for conservation.

Certain devices can specify time-out values of -1 to use the standard power policy time-outs for their device class. The standard time-out values provide for better system integration for supported standard device classes. At present, WDM supports this feature for devices of type FILE_DEVICE_DISK and FILE_DEVICE_MASS_STORAGE. **PoRegisterDeviceForIdleDetection** returns **NULL** if -1 is specified for a device of an unsupported type. (For information about device types, see Specifying Device Types.)

Only one idle detection can be set per device. Subsequent calls to **PoRegisterDeviceForIdleDetection** change the idle detection values.

If both *ConservationIdleTime* and *PerformanceIdleTime* are zero, this routine cancels all idle detection for the device and returns **NULL**.

PoRegisterDeviceForIdleDetection can free a driver from the need to perform its own idle detection. However, drivers can also implement their own idle detection.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

IRP

IRP_MN_SET_POWER

PoSetDeviceBusy

PoRegisterPowerSettingCallback function

1/25/2019 • 4 minutes to read • Edit Online

The **PoRegisterPowerSettingCallback** routine registers a power-setting callback routine to receive notifications of changes in the specified power setting.

Syntax

```
NTSTATUS PoRegisterPowerSettingCallback(
  PDEVICE_OBJECT          DeviceObject,
  LPCGUID                 SettingGuid,
  PPOWER_SETTING_CALLBACK Callback,
  PVOID                   Context,
  PVOID                   *Handle
);
```

Parameters

`DeviceObject`

A pointer to a DEVICE_OBJECT structure that is associated with the caller of this routine. This parameter is optional. It is used internally only for debugging purposes. If this parameter is not supplied, it must be set to **NULL**.

`SettingGuid`

A pointer to the GUID that represents the power setting for this registration. When the specified power setting changes, the power manager calls the callback routine to notify the driver of the change and to supply the new value of the setting. For more information, see Remarks.

`Callback`

A pointer to a caller-implemented power-setting callback routine that the power manager calls when the specified power setting changes. For the functional prototype for the callback routine, see Power-Setting Callback.

`Context`

A pointer to the context for the callback routine. This parameter is optional. It is provided so that a driver or device context can be passed to the callback routine. If this parameter is not used, it must be set to **NULL**.

`Handle`

A handle that the power manager uses to represent the callback routine. A driver must subsequently supply this handle in a call to PoUnregisterPowerSettingCallback to unregister the callback routine.

Return Value

PoRegisterPowerSettingCallback returns one of the following:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The routine registered the callback routine.
STATUS_INSUFFICIENT_RESOURCES	The routine could not allocate the system resources that are required to register the callback routine.

Remarks

A driver calls **PoRegisterPowerSettingCallback** to register a callback routine with the power manager. The power manager subsequently calls this callback routine to notify the driver after there is a change to the specified power setting. In addition, the power manager initializes the power setting of the driver by immediately calling the callback routine and passing the current value of the power setting. The power manager initializes the power setting of the driver this way regardless of whether the power setting has actually changed.

A driver should call **PoRegisterPowerSettingCallback** for each power setting that the driver needs to monitor. Drivers should call this routine in their DriverEntry routine during initialization. Typically, most drivers pass a pointer to a device extension in the *Context* parameter.

To unregister a power-setting callback, call the PoUnregisterPowerSettingCallback routine.

Typically, Kernel-Mode Driver Framework (KMDF) drivers should call **PoRegisterPowerSettingCallback** from their EvtDeviceSelfManagedIoInit callback function, and should call **PoUnregisterPowerSettingCallback** from their EvtDeviceSelfManagedIoCleanup callback function. These drivers should not call **PoRegisterPowerSettingCallback** from their EvtDriverDeviceAdd callback function; otherwise, the power-setting callback routine might be called before the driver stack is completely built.

The callback routine that is registered for a particular power setting is called when a transition in power state occurs that changes the value of the setting, or when the power manager changes the value of the setting. For example, if *SettingGuid* points to the GUID value GUID_LIDSWITCH_STATE_CHANGE, the callback routine is called when the lid to a laptop computer clicks open or closed. The *Value* parameter passed to the callback routine in this example points to a ULONG value that is 1 if the state of the lid switch changed from closed to open, and is 0 if the state of the lid switch changed from open to closed. For more information, see the power-setting GUID definitions and extensive comments in the Wdm.h header file.

The initial call to a callback routine might occur immediately, before the **PoRegisterPowerSettingCallback** call that registers the routine returns.

PoRegisterPowerSettingCallback can be called only at IRQL = PASSIVE_LEVEL.

Power-Setting Callback

The function prototype of the power-setting callback routine is as follows:

```
NTSTATUS
POWER_SETTING_CALLBACK (
  _In_ LPCGUID SettingGuid,
  _In_ PVOID Value,
  _In_ ULONG ValueLength,
  _Inout_opt_ PVOID Context
);
```

The power-setting callback parameters are:

SettingGuid
A pointer to a GUID that represents the power setting that changed. Power settings and their corresponding GUIDs are defined in Wdm.h.

Value A pointer to the new value of the power setting that changed.

ValueLength A value of type ULONG that specifies the size, in bytes, of the new power setting value.

Context
The pointer to the context that a driver supplied in the call to **PoRegisterPowerSettingCallback** that registered the callback routine.

The power manager calls a power-setting callback at IRQL = PASSIVE_LEVEL. #### Examples To define a power-setting callback routine, you must first provide a function declaration that identifies the type of callback function you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system. For example, to define a power-setting callback routine that is named `MyPowerSettingCallback`, use the POWER_SETTING_CALLBACK type as shown in this code example:

```
POWER_SETTING_CALLBACK MyPowerSettingCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyPowerSettingCallback(
    LPCGUID SettingGuid,
    PVOID Value,
    ULONG ValueLength,
    PVOID Context
    )
  {
      // Function body
  }
```

The POWER_SETTING_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the POWER_SETTING_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For

information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)

See Also

DriverEntry

EvtDeviceSelfManagedIoCleanup

EvtDeviceSelfManagedIoInit

EvtDriverDeviceAdd

PoUnregisterPowerSettingCallback

PoRegisterSystemState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoRegisterSystemState** routine registers the system as busy due to certain activity.

Syntax

```
PVOID PoRegisterSystemState(
  PVOID           StateHandle,
  EXECUTION_STATE Flags
);
```

Parameters

`StateHandle`

A pointer to a caller-supplied buffer for a registration state handle. The size, in bytes, of the buffer is **sizeof**(ULONG). If **NULL**, this is a new registration. If non-**NULL**, this parameter points to a handle that was returned by a previous call to **PoRegisterSystemState**.

`Flags`

Indicates the type of activity, as specified by a bitwise OR of one or more of the following values:

ES_SYSTEM_REQUIRED

The system is not idle, regardless of apparent load.

ES_DISPLAY_REQUIRED

Use of the display is required.

ES_USER_PRESENT

A user is present.

ES_CONTINUOUS

The settings are continuous and should remain in effect until explicitly changed.

Return Value

PoRegisterSystemState returns a handle to be used later to change or unregister the system busy state. It returns **NULL** if the handle could not be allocated.

Remarks

PoRegisterSystemState registers the system busy state as indicated by the flags. The registration persists until the caller explicitly changes it with another call to **PoRegisterSystemState** or cancels it with a call to PoUnregisterSystemState.

The *Flags* parameter specifies the type of activity in progress. Drivers can specify any combination of the flags.

Setting ES_CONTINUOUS makes the busy state persist until a driver explicitly changes or cancels it by calling **PoRegisterSystemState** or **PoUnregisterSystemState**.

A driver can set the system busy state to request that the power manager avoid system power state transitions out

of the system working state (S0) while driver activity is occurring. Note, however, that under some circumstances (such as a critically low battery) the power manager may override this request and put the system to sleep anyway.

To set the system power state, call PoSetSystemState.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoSetSystemState

PoUnregisterSystemState

PoRequestPowerIrp function

1/25/2019 • 3 minutes to read • Edit Online

The **PoRequestPowerIrp** routine allocates a power IRP and sends it to the top driver in the device stack for the specified device.

Syntax

```
NTSTATUS PoRequestPowerIrp(
  PDEVICE_OBJECT          DeviceObject,
  UCHAR                   MinorFunction,
  POWER_STATE             PowerState,
  PREQUEST_POWER_COMPLETE CompletionFunction,
  __drv_aliasesMem PVOID  Context,
  PIRP                    *Irp
);
```

Parameters

`DeviceObject`

A pointer to the target DEVICE_OBJECT for the IRP. In Windows 2000 and later versions of Windows, this parameter can point to a physical device object (PDO) or a functional device object (FDO). In Windows 98/Me, this parameter must point to the PDO of the underlying device.

`MinorFunction`

Specifies one of the following minor power IRP codes: IRP_MN_QUERY_POWER, IRP_MN_SET_POWER, or IRP_MN_WAIT_WAKE.

`PowerState`

Specifies a POWER_STATE value to pass in the IRP. For **IRP_MN_SET_POWER** and **IRP_MN_QUERY_POWER**, specify the requested new device power state. Possible values are DEVICE_POWER_STATE values.

For **IRP_MN_WAIT_WAKE**, specify the lowest (least-powered) system power state from which the device should be allowed to wake the system. Possible values are SYSTEM_POWER_STATE values.

`CompletionFunction`

A pointer to the caller's PowerCompletion callback routine. The I/O manager calls this routine when the IRP has completed. This parameter is optional and can be set to **NULL** if no *PowerCompletion* callback routine is needed.

`Context`

A pointer to a caller-supplied context to be passed through to the *PowerCompletion* callback. When the caller requests a device set-power IRP in response to a system set-power IRP, *Context* should contain the system set-power IRP that triggered the request.

`Irp`

A pointer to a caller-supplied variable in which **PoRequestPowerIrp** returns a pointer to the IRP that it allocates. Specify a value for *Irp* only if the *MinorFunction* parameter is set to **IRP_MN_WAIT_WAKE**. Otherwise, this parameter should always be **NULL** because the IRP might be completed before **PoRequestPowerIrp** returns,

causing this parameter to point to memory that has already been discarded.

Return Value

PoRequestPowerIrp returns one of the following:

RETURN CODE	DESCRIPTION
STATUS_PENDING	The IRP has been sent.
STATUS_INSUFFICIENT_RESOURCES	The routine could not allocate the IRP.
STATUS_INVALID_PARAMETER_2	*MinorFunction* does not signify a valid minor power IRP code.

Remarks

A device power policy owner calls this routine to send a wait/wake, query, or set-power IRP.

A driver calls **PoRequestPowerIrp**—not IoAllocateIrp—to allocate and send a power IRP that has minor IRP code **IRP_MN_SET_POWER**, **IRP_MN_QUERY_POWER**, or **IRP_MN_WAIT_WAKE**. (A driver must call **IoAllocateIrp** to send a power IRP with minor IRP code IRP_MN_POWER_SEQUENCE.)

If **PoRequestPowerIrp** returns a status value of STATUS_PENDING, the routine successfully allocated a device power IRP and sent it to the top of the device stack for the device. After the bus driver and all other drivers have completed the IRP, and the I/O manager has called all IoCompletion routines set by drivers as they passed the IRP down the device stack, the I/O manager calls the PowerCompletion routine and passes to this routine the specified *Context* value. If **PoRequestPowerIrp** returns a status other than STATUS_PENDING, the routine did not send a device power IRP and the *PowerCompletion* routine is not called.

The *PowerCompletion* routine performs any additional tasks the sender of the IRP requires after all other drivers have completed the IRP. It need not free the IRP; the power manager does that. In Windows 2000 and later versions of Windows, the *PowerCompletion* routine can be called at IRQL = PASSIVE_LEVEL or IRQL = DISPATCH_LEVEL. In Windows 98/Me, the *PowerCompletion* routine is always called at IRQL = PASSIVE_LEVEL, and drivers must complete IRPs at IRQL = PASSIVE_LEVEL.

A device power policy owner calls **PoRequestPowerIrp** to send a device query- or set-power IRP when it receives a system query- or set-power IRP. The driver should set an *IoCompletion* routine in the system IRP and pass the system IRP to the next lower driver. The *IoCompletion* routine calls **PoRequestPowerIrp** to send the device IRP, passing the system IRP in the *Context* parameter. The *Context* parameter is subsequently passed to the *PowerCompletion* routine for the device IRP. In the *PowerCompletion* routine, the driver can complete the system IRP. For more information, see Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States and Wait/Wake Callback Routines.

Drivers can use the returned *Irp* to cancel an **IRP_MN_WAIT_WAKE** IRP. Drivers that are requesting other types of power IRPs must pass **NULL** for this parameter.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	MarkDevicePower, PowerDownFail, PowerUpFail, RequestedPowerIrp, HwStorPortProhibitedDDIs

See Also

IO_STATUS_BLOCK

IRP

IRP_MN_POWER_SEQUENCE

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IRP_MN_WAIT_WAKE

IoAllocateIrp

PoStartNextPowerIrp

PowerCompletion

PoSetDeviceBusyEx function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetDeviceBusyEx** routine notifies the power manager that the device associated with the specified idle counter is busy.

Syntax

```
void PoSetDeviceBusyEx(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoSetDeviceBusyEx**.

Return Value

None

Remarks

This routine is a direct replacement for the PoSetDeviceBusy macro. If you are writing new driver code for Windows Vista with Service Pack 1 (SP1) and later versions of Windows, call **PoSetDeviceBusyEx** instead of **PoSetDeviceBusy**.

A driver calls the **PoSetDeviceBusyEx** and **PoRegisterDeviceForIdleDetection** routines to enable system idle detection for its device. If a device that is registered for idle detection stays idle for the driver-specified time-out period, the power manager sends an IRP_MN_SET_POWER request to put the device in a requested sleep state.

PoSetDeviceBusyEx reports that the device is busy, so that the power manager can restart its idle countdown. If the device is in a sleep state, **PoSetDeviceBusyEx** does not change the state of the device. That is, it does not cause the system to send an **IRP_MN_SET_POWER** request to awaken the device.

PoSetDeviceBusyEx is designed for use with I/O operations that are relatively brief compared to the time-out period of the idle counter. For longer operations that might exceed this period, use the PoStartDeviceBusy and PoEndDeviceBusy routines instead.

A driver that makes multiple requests for brief I/O operations should call **PoSetDeviceBusyEx** for every I/O request that it makes.

Requirements

Minimum supported client	Available starting with Windows Vista with SP1.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IRP_MN_SET_POWER

PoEndDeviceBusy

PoRegisterDeviceForIdleDetection

PoSetDeviceBusy

PoStartDeviceBusy

PoSetPowerRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetPowerRequest** routine increments the count for the specified power request type.

Syntax

```
NTSTATUS PoSetPowerRequest(
  PVOID             PowerRequest,
  POWER_REQUEST_TYPE Type
);
```

Parameters

`PowerRequest`

A pointer to a power request object that was created by the PoCreatePowerRequest routine.

`Type`

The type of power request. Set this parameter to the following POWER_REQUEST_TYPE enumeration value:

- **PowerRequestSystemRequired**

Return Value

PoSetPowerRequest returns STATUS_SUCCESS if the call is successful. If the call fails, possible error return codes include the following:

RETURN CODE	DESCRIPTION
STATUS_UNSUPPORTED_TYPE	The *Type* parameter is set to an unsupported value.

Remarks

A driver can call the **PoSetPowerRequest** routine to request that the power manager override several types of default power behavior, which are specified as POWER_REQUEST_TYPE enumeration values. To restore the default behavior, the driver cancels the request by calling the PoClearPowerRequest routine.

The power manager maintains a count of the active requests for each power request type. The **PoSetPowerRequest** routine increments the count for the specified power request type by one. The **PoClearPowerRequest** routine decrements the count by one. A nonzero count indicates that requests from one or more components are active. After the count decrements to zero, the computer reverts to the default behavior for the specified power request type.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

POWER_REQUEST_TYPE

PoClearPowerRequest

PoCreatePowerRequest

PoSetPowerState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetPowerState** routine notifies the system of a change in the device power state for a device.

Syntax

```
POWER_STATE PoSetPowerState(
  PDEVICE_OBJECT   DeviceObject,
  POWER_STATE_TYPE Type,
  POWER_STATE      State
);
```

Parameters

`DeviceObject`

A pointer to the target DEVICE_OBJECT.

`Type`

Indicates a POWER_STATE_TYPE value. Drivers must specify **DevicePowerState**.

`State`

Specifies the power state to be set. Drivers must specify a DEVICE_POWER_STATE value.

Return Value

On Windows 2000 and later versions of the operating system, **PoSetPowerState** returns the previous power state. On Windows 98/Me, **PoSetPowerState** returns the state passed in *State*.

Remarks

PoSetPowerState notifies the power manager of the new power state for a device. Each driver in a device stack (filter, function, and bus drivers) must call **PoSetPowerState** to inform the power manager of a change in the power state of its corresponding device object. For example:

- A driver calls this routine after receiving a device set-power request and before calling PoStartNextPowerIrp. When handling a PnP IRP_MN_START_DEVICE request, the driver should call **PoSetPowerState** to notify the power manager that the device is in the D0 state.
- If the device is powering down, the driver must call **PoSetPowerState** before leaving the D0 state. In addition, the driver must be able to process client requests before **PoSetPowerState** returns.
- If the device is powering up, the driver must call **PoSetPowerState** after the device is successfully put into the D0 state.

For more information about powering a device up and down, see Handling IRP_MN_SET_POWER for Device Power States.

Callers of **PoSetPowerState** must be running at IRQL <= APC_LEVEL except when setting state to D0. When setting state to D0, callers can be running at IRQL <= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

IRP_MN_START_DEVICE

PoStartNextPowerIrp

PoSetSystemState function

1/25/2019 • 2 minutes to read • Edit Online

Drivers call the **PoSetSystemState** routine to indicate that the system is active.

Syntax

```
void PoSetSystemState(
  EXECUTION_STATE Flags
);
```

Parameters

`Flags`

Indicates the system activity, as specified by a bitwise OR of one or more of the following values:

ES_SYSTEM_REQUIRED

The system is not idle, regardless of apparent load.

ES_DISPLAY_REQUIRED

Use of the display is required.

ES_USER_PRESENT

A user is present.

Return Value

None

Remarks

A driver calls **PoSetSystemState** to set flags indicating that system activity is occurring. Unlike PoRegisterSystemState, this routine does not allow the driver to set a persistent busy state.

The *Flags* parameter specifies the type of activity occurring. Drivers can specify any combination of the flags.

Drivers can set the system busy state to request that the system avoid leaving of the working state while driver activity is occurring. Note, however, that under some circumstances (such as a critically low battery) the power manager may override this request and put the system to sleep anyway.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

PoRegisterSystemState

PoUnregisterSystemState

PoSetSystemWake function

1/25/2019 • 2 minutes to read • Edit Online

The **PoSetSystemWake** routine marks the specified IRP as one that contributed to waking the system from a sleep state.

Syntax

```
void PoSetSystemWake(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to the wait/wake IRP.

Return Value

None

Remarks

Drivers call **PoSetSystemWake** to mark an IRP_MN_WAIT_WAKE IRP as contributing to waking the system from a sleep state. By default, wait/wake IRPs are considered to be device wake-up IRPs. It is the responsibility of the terminal device in a wait/wake chain to determine if it woke the system and to call **PoSetSystemWake** for the terminal wait/wake IRP. When a driver calls **PoSetSystemWake** on an IRP, it is marked as having contributed to waking the system from a sleep state. Only one driver in a stack needs to call this routine, and it should normally be the bus driver in a driver stack.

All other drivers in a wait/wake chain can call PoGetSystemWake for their own wait/wake IRPs at completion to determine if they should call **PoSetSystemWake** on any child wait/wake IRPs that they are about to complete. This ensures that system wake information properly progresses throughout the wait/wake chain.

After a wait/wake IRP completes, the power manager checks if the IRP is marked as a system wake IRP. If the IRP is marked as a system wake IRP, the power manager adds the IRP to an internal list of the devices that woke the system. However, the power manager only keeps track of the most specific devices that work the system. For example, if device A is added as a device that woke the system, and then device B—a child of device A—is also added, the power manager only retains device B in the list because device B is the most specific. If the power manager cannot determine the most specific device that woke the system, the power manager might keep track of more than one device that reported it woke the system.

The power manager logs an Event Tracing for Windows (ETW) event (viewable in the global system channel) that includes information about which devices woke the system.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

PoGetSystemWake

PoStartDeviceBusy function

1/25/2019 • 2 minutes to read • Edit Online

The **PoStartDeviceBusy** routine marks the start of a period of time in which the device is busy.

Syntax

```
void PoStartDeviceBusy(
  PULONG IdlePointer
);
```

Parameters

`IdlePointer`

A pointer to an idle counter. This is a pointer value that was previously returned by the PoRegisterDeviceForIdleDetection routine. Because **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer, the caller must verify that the pointer is non-**NULL** before it calls **PoStartDeviceBusy**.

Return Value

None

Remarks

The **PoStartDeviceBusy** and PoEndDeviceBusy routines mark the start and end of a time period in which a device is busy. Each call to **PoStartDeviceBusy** must be followed by a corresponding call to **PoEndDeviceBusy**.

For each device, the power manager maintains a count of the number of outstanding **PoStartDeviceBusy** calls for which it has not yet received a corresponding **PoEndDeviceBusy** call. A **PoStartDeviceBusy** call increments the busy count by one. A **PoEndDeviceBusy** call decrements the busy count by one. A nonzero busy count disables the idle counter for the device. After the busy count reaches zero, the power manager resets the idle counter to the time-out period and enables the counter.

Rather than use the **PoStartDeviceBusy** and **PoEndDeviceBusy** routines to reset the idle counter, you can call the PoSetDeviceBusyEx routine (or the PoSetDeviceBusy macro). To disable the idle counter during an extended busy period, a driver can call the **PoRegisterDeviceForIdleDetection** routine to disable and enable idle notifications at the start and end of the busy period. However, **PoStartDeviceBusy** and **PoEndDeviceBusy** are typically more convenient to use for this purpose, and you should consider using these routines in new code that you write for Windows 7 and later versions of Windows.

Requirements

Minimum supported client	Available starting with Windows 7.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

PoEndDeviceBusy

PoRegisterDeviceForIdleDetection

PoSetDeviceBusy

PoStartNextPowerIrp function

1/25/2019 • 2 minutes to read • Edit Online

The **PoStartNextPowerIrp** routine signals the power manager that the driver is ready to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)

Syntax

```
void PoStartNextPowerIrp(
  PIRP Irp
);
```

Parameters

`Irp`

A pointer to an IRP in which the major function code is IRP_MJ_POWER.

Return Value

None

Remarks

Starting with Windows Vista, the driver is not required to call **PoStartNextPowerIrp** and a call to this routine does not perform a power management operation. However, on Windows Server 2003, Windows XP, and Windows 2000, **PoStartNextPowerIrp** must be called by every driver in a device stack after the driver is finished with the previous power IRP, if any, and is ready to handle the next power IRP. It must be called once by each driver for every IRP_MN_QUERY_POWER or IRP_MN_SET_POWER request.

Although power IRPs are completed only once, typically by the bus driver for a device, each driver in the device stack must call **PoStartNextPowerIrp** as the IRP travels down or back up the stack. Even if a driver fails the IRP, the driver must nevertheless call **PoStartNextPowerIrp** to signal the power manager that it is ready to handle another power IRP.

The driver must call **PoStartNextPowerIrp** while the current IRP stack location points to the current driver. Therefore, this routine must be called before IoCompleteRequest, IoSkipCurrentIrpStackLocation, and PoCallDriver. As a general rule, a driver should call **PoStartNextPowerIrp** from its *IoCompletion* routine associated with the IRP or from the callback routine it passed to PoRequestPowerIrp.

Bus drivers must call **PoStartNextPowerIrp** before completing each IRP.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=DISPATCH_LEVEL"

See Also

IRP

IRP_MJ_POWER

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IoCompleteRequest

IoSkipCurrentIrpStackLocation

PoCallDriver

PoRequestPowerIrp

PoUnregisterPowerSettingCallback function

1/25/2019 • 2 minutes to read • Edit Online

The **PoUnregisterPowerSettingCallback** routine unregisters a power-setting callback routine that a driver previously registered by calling the PoRegisterPowerSettingCallback routine.

Syntax

```
NTSTATUS PoUnregisterPowerSettingCallback(
  PVOID Handle
);
```

Parameters

`Handle`

A handle to a callback routine that a driver registered by calling **PoRegisterPowerSettingCallback**.

Return Value

PoUnregisterPowerSettingCallback returns one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The callback routine was unregistered.
STATUS_INVALID_PARAMETER	The *Handle* value is not a valid handle to a power setting callback routine.

Remarks

A driver calls **PoUnregisterPowerSettingCallback** to unregister a power setting callback routine that the driver previously registered by calling **PoRegisterPowerSettingCallback**.

A driver must call **PoUnregisterPowerSettingCallback** to unregister each callback routine that it previously registered. All callback routines registered by a driver should be unregistered in the *Unload* routine of the driver.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

PoRegisterPowerSettingCallback

PoUnregisterSystemState function

1/25/2019 • 2 minutes to read • Edit Online

The **PoUnregisterSystemState** routine cancels a system state registration created by PoRegisterSystemState.

Syntax

```
void PoUnregisterSystemState(
  PVOID StateHandle
);
```

Parameters

`StateHandle`

A pointer to a handle previously returned by **PoRegisterSystemState**.

Return Value

None

Remarks

This routine cancels a system busy state registration established by **PoRegisterSystemState** and frees the associated *StateHandle*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

PoRegisterSystemState

PPUT_DMA_ADAPTER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **PutDmaAdapter** routine frees a DMA_ADAPTER structure previously allocated by IoGetDmaAdapter.

Syntax

```
PPUT_DMA_ADAPTER PputDmaAdapter;

void PputDmaAdapter(
  PDMA_ADAPTER DmaAdapter
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure to be released.

Return Value

None

Remarks

PutDmaAdapter is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

PutDmaAdapter frees a DMA adapter object previously allocated by **IoGetDmaAdapter**. Drivers should call **PutDmaAdapter** after completing DMA operations and freeing any map registers and common buffer allocated with this adapter object. After **PutDmaAdapter** returns, the driver can no longer use the DMA adapter object.

A driver must call **PutDmaAdapter** when it receives a PnP IRP_MN_STOP_DEVICE request.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

DMA_ADAPTER

DMA_OPERATIONS

IoGetDmaAdapter

PPUT_SCATTER_GATHER_LIST callback function

1/25/2019 • 2 minutes to read • Edit Online

The **PutScatterGatherList** routine frees the previously allocated map registers and scatter/gather list used in scatter/gather DMA.

Syntax

```
PPUT_SCATTER_GATHER_LIST PputScatterGatherList;

void PputScatterGatherList(
  PDMA_ADAPTER DmaAdapter,
  PSCATTER_GATHER_LIST ScatterGather,
  BOOLEAN WriteToDevice
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the DMA_ADAPTER structure returned by IoGetDmaAdapter that represents the bus-master adapter or DMA controller.

`ScatterGather`

Pointer to a SCATTER_GATHER_LIST structure previously returned by GetScatterGatherList.

`WriteToDevice`

Indicates the direction of the DMA transfer: specify **TRUE** for a transfer from the buffer to the device, and **FALSE** otherwise.

Return Value

None

Remarks

PutScatterGatherList is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

Drivers should call **PutScatterGatherList** after completing scatter/gather I/O. This routine flushes the adapter buffers, frees the map registers, and unmaps and frees the associated MDLs. Additionally, the routine frees the scatter/gather list if it was previously allocated by GetScatterGatherList.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows. Not supported in Windows 98 or Windows Me.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	DISPATCH_LEVEL
DDI compliance rules	IrqlDispatch, IrqlDispatch(storport)

See Also

DMA_ADAPTER

DMA_OPERATIONS

GetScatterGatherList

IoGetDmaAdapter

SCATTER_GATHER_LIST

PREAD_DMA_COUNTER callback function

1/25/2019 • 2 minutes to read • Edit Online

The **ReadDmaCounter** routine returns the number of bytes remaining to be transferred during the current subordinate DMA operation.

Syntax

```
PREAD_DMA_COUNTER PreadDmaCounter;

ULONG PreadDmaCounter(
  PDMA_ADAPTER DmaAdapter
)
{...}
```

Parameters

`DmaAdapter`

Pointer to the adapter object previously returned by IoGetDmaAdapter representing the system DMA controller channel currently in use.

Return Value

ReadDmaCounter returns the number of bytes remaining to be transferred in the current DMA operation.

Remarks

ReadDmaCounter is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling IoGetDmaAdapter.

ReadDmaCounter can be called only by drivers of subordinate DMA devices. Usually, the caller is the driver of a subordinate device that uses a system DMA controller's autoinitialize mode.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= DISPATCH_LEVEL"
DDI compliance rules	IrqlDispatch

See Also

AllocateCommonBuffer

DMA_ADAPTER

FlushAdapterBuffers

IoGetDmaAdapter

MapTransfer

PREENUMERATE_SELF callback function

1/25/2019 • 2 minutes to read • Edit Online

A *ReenumerateSelf* routine requests that a bus driver reenumerate a child device.

Syntax

```
PREENUMERATE_SELF PreenumerateSelf;

void PreenumerateSelf(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the REENUMERATE_SELF_INTERFACE_STANDARD structure.

Return Value

None

Remarks

The REENUMERATE_SELF_INTERFACE_STANDARD interface structure supplies a pointer to the *ReenumerateSelf* routine.

If the bus driver can process the request at the time of the routine call, the bus driver asynchronously reports the device as missing in a subsequent IRP_MN_QUERY_DEVICE_RELATIONS BusRelations query. This report causes the current device instance to receive IRP_MN_SURPRISE_REMOVAL and IRP_MN_REMOVE_DEVICE requests. After the surprise removal has been processed, a new instance of the device is enumerated and the driver's AddDevice routine is called to rebuild the device stack.

Requirements

Minimum supported client	Available in Windows Vista.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h)
IRQL	PASSIVE_LEVEL

See Also

IRP_MN_QUERY_DEVICE_RELATIONS

IRP_MN_REMOVE_DEVICE

IRP_MN_SURPRISE_REMOVAL

REENUMERATE_SELF_INTERFACE_STANDARD

ProbeForRead function

1/25/2019 • 2 minutes to read • Edit Online

The **ProbeForRead** routine checks that a user-mode buffer actually resides in the user portion of the address space, and is correctly aligned.

Syntax

```
void ProbeForRead(
  const volatile VOID *Address,
  SIZE_T              Length,
  ULONG               Alignment
);
```

Parameters

`Address`

Specifies the beginning of the user-mode buffer.

`Length`

Specifies the length, in bytes, of the user-mode buffer.

`Alignment`

Specifies the required alignment, in bytes, of the beginning of the user-mode buffer.

Return Value

None

Remarks

If the specified range of memory is not within the user-mode address range, **ProbeForRead** raises the STATUS_ACCESS_VIOLATION exception. If the beginning of the address range is not aligned on the byte boundary that is specified by *Alignment*, **ProbeForRead** raises the STATUS_DATATYPE_MISALIGNMENT exception.

Kernel-mode drivers must use **ProbeForRead** to validate read access to buffers that are allocated in user space. It is most commonly used during METHOD_NEITHER I/O to validate the user buffer pointed to by **Irp -> UserBuffer**.

Drivers must call **ProbeForRead** inside a **try/except** block. If the routine raises an exception, the driver should complete the IRP with the appropriate error. Note that subsequent accesses by the driver to the user-mode buffer must also be encapsulated within a **try/except** block: a malicious application could have another thread deleting, substituting, or changing the protection of user address ranges at any time (even after or during a call to **ProbeForRead** or **ProbeForWrite**). For more information, see Handling Exceptions.

Do not use this routine on kernel-mode addresses; it will raise an exception.

If **Irp->RequestorMode** = **KernelMode**, the **Irp->AssociatedIrp.SystemBuffer** and **Irp->UserBuffer** fields do not contain user-mode addresses, and a call to **ProbeForRead** to probe a buffer pointed to by either field will raise

an exception.

If *Length* = 0, **ProbeForRead** does no checking of the address. In this case, the routine does not raise an exception for an address that is misaligned or is outside the range of valid user addresses.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ProbeForWrite

ProbeForWrite function

1/25/2019 • 2 minutes to read • Edit Online

The **ProbeForWrite** routine checks that a user-mode buffer actually resides in the user-mode portion of the address space, is writable, and is correctly aligned.

Syntax

```
void ProbeForWrite(
  volatile VOID *Address,
  SIZE_T        Length,
  ULONG         Alignment
);
```

Parameters

`Address`

Specifies the beginning of the user-mode buffer.

`Length`

Specifies the length, in bytes, of the user-mode buffer.

`Alignment`

Specifies the required alignment, in bytes, of the beginning of the user-mode buffer.

Return Value

None

Remarks

If the specified range of memory is not a valid user-mode address range or is not writable (no access, read-only, and so on), **ProbeForWrite** raises the STATUS_ACCESS_VIOLATION exception. If the beginning of the address range is not aligned on the byte boundary that is specified by *Alignment*, **ProbeForWrite** raises the STATUS_DATATYPE_MISALIGNMENT exception.

Kernel-mode drivers must use **ProbeForWrite** to validate write access to buffers allocated in user space. It is most commonly used during METHOD_NEITHER I/O to validate the user buffer pointed to by **Irp->UserBuffer**.

Drivers must call **ProbeForWrite** inside a **try/except** block. If the routine raises an exception, the driver should complete the IRP with the appropriate error. Note that subsequent accesses by the driver to the user-mode buffer must also be encapsulated within a **try/except** block: a malicious application could have another thread deleting, substituting, or changing the protection of user address ranges at any time (even after or during a call to **ProbeForRead** or **ProbeForWrite**). For more information, see Handling Exceptions.

Do not use this routine on kernel-mode addresses; it will raise an exception.

If **Irp->RequestorMode** = **KernelMode**, the **Irp->AssociatedIrp.SystemBuffer** and **Irp->UserBuffer** fields do not contain user-mode addresses, and a call to **ProbeForWrite** to probe a buffer pointed to by either field will raise an exception.

If *Length* = 0, **ProbeForWrite** does no checking of the address. In this case, the routine does not raise an exception for an address that is misaligned or is outside the range of valid user addresses.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"
DDI compliance rules	IrqlExApcLte2, HwStorPortProhibitedDDIs

See Also

ProbeForRead

PROCESSOR_HALT_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

A *Halt* callback routine transitions the processor to an idle state.

Syntax

```
PROCESSOR_HALT_ROUTINE ProcessorHaltRoutine;

NTSTATUS ProcessorHaltRoutine(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to a PEP-defined processor-halt context. This pointer is the *Context* parameter value that the PEP previously passed to the ProcessorHalt routine.

Return Value

A *Halt* callback routine may or may not return. If this routine does return, it returns STATUS_SUCCESS to indicates that the processor successfully entered the idle state. Otherwise, it returns an appropriate error status code.

Remarks

This routine is implemented by the platform extension plug-in (PEP) and is called by the Windows power management framework (PoFx). The ProcessorHalt routine accepts a pointer to a *Halt* callback routine as a parameter.

The PEP's *Halt* routine is called at the same IRQL at which the PEP called **ProcessorHalt**.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	wdm.h (include Pepfx.h)
IRQL	See Remarks.

See Also

PEP_CRASHDUMP_INFORMATION

ProcessorHalt

PsCreateSystemThread function

1/25/2019 • 3 minutes to read • Edit Online

The **PsCreateSystemThread** routine creates a system thread that executes in kernel mode and returns a handle for the thread.

Syntax

```
NTSTATUS PsCreateSystemThread(
  PHANDLE            ThreadHandle,
  ULONG              DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  HANDLE             ProcessHandle,
  PCLIENT_ID         ClientId,
  PKSTART_ROUTINE    StartRoutine,
  PVOID              StartContext
);
```

Parameters

`ThreadHandle`

Points to a variable that will receive the handle. The driver must close the handle with ZwClose once the handle is no longer in use. This handle is a kernel handle for Windows Vista and later versions of Windows. In earlier versions of Windows, the handle might not be a kernel handle.

`DesiredAccess`

Specifies the ACCESS_MASK value that represents the requested types of access to the created thread.

`ObjectAttributes`

Points to a structure that specifies the object's attributes. OBJ_PERMANENT, OBJ_EXCLUSIVE, and OBJ_OPENIF are not valid attributes for a thread object. On Windows XP and later versions of Windows, if the caller is not running in the system process context, it must set the OBJ_KERNEL_HANDLE attribute for *ObjectAttributes*. Drivers for Microsoft Windows 2000 and Windows 98/Me must only call **PsCreateSystemThread** from the system process context. For Windows Vista and later versions of Windows, the handle will be a kernel handle.

`ProcessHandle`

Specifies an open handle for the process in whose address space the thread is to be run. The caller's thread must have PROCESS_CREATE_THREAD access to this process. If this parameter is not supplied, the thread will be created in the initial system process. This value should be **NULL** for a driver-created thread. Use the **NtCurrentProcess** macro, defined in Ntddk.h, to specify the current process.

`ClientId`

Points to a structure that receives the client identifier of the new thread. This value should be **NULL** for a driver-created thread.

`StartRoutine`

The entry point for the newly created system thread. This parameter is a function pointer to a ThreadStart routine that receives a single argument, which is the *StartContext* parameter value supplied by the caller.

`StartContext`

Supplies a single argument that is passed to the thread when it begins execution.

Return Value

PsCreateSystemThread returns STATUS_SUCCESS if the thread was created.

Remarks

Drivers that create device-dedicated threads call this routine, either when they initialize or when I/O requests begin to come in to such a driver's Dispatch routines. For example, a driver might create such a thread when it receives an asynchronous device control request.

PsCreateSystemThread creates a kernel-mode thread that begins a separate thread of execution within the system. Such a system thread has no TEB or user-mode context and runs only in kernel mode.

If the input *ProcessHandle* is **NULL**, the created thread is associated with the system process. Such a thread continues running until either the system is shut down or the thread terminates itself by calling **PsTerminateSystemThread**.

Starting with Windows XP, driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE attribute for the *ObjectAttributes* parameter of **PsCreateSystemThread**. This restricts the use of the handle returned by **PsCreateSystemThread** to processes running in kernel mode. Otherwise, the thread handle can be accessed by the process in whose context the driver is running. Drivers can set the OBJ_KERNEL_HANDLE attribute as follows.

```
InitializeObjectAttributes(&ObjectAttributes, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
```

Drivers for Windows 2000 and Windows 98/Me must call **PsCreateSystemThread** only from the system process context.

For more information about the *StartContext* parameter, see ThreadStart.

The newly created system thread runs at PASSIVE_LEVEL inside a critical region with normal kernel APCs disabled.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	IrqlPsPassive, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

KeSetBasePriorityThread

KeSetPriorityThread

PsTerminateSystemThread

ThreadStart

ZwSetInformationThread

PsGetCurrentThread function

1/25/2019 • 2 minutes to read • Edit Online

The **PsGetCurrentThread** routine identifies the current thread.

Syntax

```
PETHREAD PsGetCurrentThread(

);
```

Parameters

This function has no parameters.

Return Value

PsGetCurrentThread returns a pointer to the executive thread object that represents the currently executing thread.

Remarks

This macro

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

KeGetCurrentThread

PsCreateSystemThread

PsGetCurrentProcess

PsGetCurrentProcessId

PsIsSystemThread

PsGetVersion function

1/25/2019 • 2 minutes to read • Edit Online

This function is obsolete in Windows XP and later versions of the Windows operating system. Use RtlGetVersion instead.

PsGetVersion returns caller-selected information about the current version of the NT-based operating system.

Syntax

```
BOOLEAN PsGetVersion(
  PULONG          MajorVersion,
  PULONG          MinorVersion,
  PULONG          BuildNumber,
  PUNICODE_STRING CSDVersion
);
```

Parameters

`MajorVersion`

Points to a caller-supplied variable that this routine sets to the major version of the operating system. This optional parameter can be **NULL**.

`MinorVersion`

Points to a caller-supplied variable that this routine sets to the minor version of the operating system. This optional parameter can be **NULL**.

`BuildNumber`

Points to a caller-supplied variable that this routine sets to the current build number of the operating system. This optional parameter can be **NULL**.

`CSDVersion`

Points to a caller-allocated buffer in which this routine returns the current service-pack version as a Unicode string only during system driver initialization. This optional parameter can be **NULL**.

Return Value

PsGetVersion returns whether the system is a checked or free build, as follows:

RETURN CODE	DESCRIPTION
TRUE (1)	Checked build of the operating system.

FAL SE (0)	Free build of the operating system.

Remarks

PsGetVersion returns the requested information, depending on which optional parameter(s) the caller supplies.

To retrieve the current service-pack number, it is easier and more efficient to make an application-level call within the Win32 environment than to call **PsGetVersion** during system driver initialization, which then must parse the string it returns at *CSDVersion*. When the registry is initialized, a driver cannot obtain this string from **PsGetVersion**, but must read the **CmCSDVersionString** value from the registry.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlPsPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

PsQueryTotalCycleTimeProcess function

1/25/2019 • 2 minutes to read • Edit Online

The **PsQueryTotalCycleTimeProcess** routine returns the accumulated cycle time for the specified process.

Syntax

```
ULONG64 PsQueryTotalCycleTimeProcess(
  PEPROCESS Process,
  PULONG64  CycleTimeStamp
);
```

Parameters

`Process`

A pointer to an EPROCESS structure that serves as the kernel object for the process.

`CycleTimeStamp`

A pointer to a ULONG64 variable to which the routine writes the current cycle counter value for the process.

Return Value

PsQueryTotalCycleTimeProcess returns the accumulated cycle time for the specified process.

Remarks

This routine uses the time stamp counter to get the number of processor clock cycles used by the specified process.

Requirements

Minimum supported client	Available in Windows 8 and later versions of Windows.
Target Platform	Universal
Header	wdm.h
Library	Ntoskrnl.lib
IRQL	PASSIVE_LEVEL

See Also

EPROCESS

PsTerminateSystemThread function

1/25/2019 • 2 minutes to read • Edit Online

The **PsTerminateSystemThread** routine terminates the current system thread.

Syntax

```
NTSTATUS PsTerminateSystemThread(
  NTSTATUS ExitStatus
);
```

Parameters

`ExitStatus`

Specifies the status of the terminating system thread.

Return Value

PsTerminateSystemThread does not return if it successfully terminates the calling thread. If the routine cannot terminate the thread (for example, if the thread is not a system thread), the routine returns an error NTSTATUS value.

Remarks

A system thread calls **PsTerminateSystemThread** to terminate itself. A driver that creates its own threads must ensure that each such thread terminates. The driver must not terminate any threads that the system or other drivers created.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlPsPassive, PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

PsCreateSystemThread

PTM_RM_NOTIFICATION callback function

1/25/2019 • 2 minutes to read • Edit Online

A resource manager's *ResourceManagerNotification* callback routine receives and handles transaction notifications.

Syntax

```
PTM_RM_NOTIFICATION PtmRmNotification;

NTSTATUS PtmRmNotification(
  PKENLISTMENT EnlistmentObject,
  PVOID RMContext,
  PVOID TransactionContext,
  ULONG TransactionNotification,
  PLARGE_INTEGER TmVirtualClock,
  ULONG ArgumentLength,
  PVOID Argument
)
{...}
```

Parameters

`EnlistmentObject`

A pointer to an enlistment object. The *ResourceManagerNotification* callback routine is receiving a notification for the enlistment that this object represents.

`RMContext`

The value that the resource manager previously specified for the *RMKey* parameter of the TmEnableCallbacks routine.

`TransactionContext`

The value that the resource manager previously specified for the *EnlistmentKey* parameter of the ZwCreateEnlistment routine.

`TransactionNotification`

One of the TRANSACTION_NOTIFY_*XXX* values that are defined in Ktmtypes.h. This value specifies the type of transaction notification that KTM sent to the caller.

`TmVirtualClock`

A pointer to a location that contains the virtual clock value at the time that KTM prepared the notification for delivery to the resource manager. If the callback routine increases this value before it returns, KTM updates the virtual clock to the new value. (Typically, resource managers do not modify the virtual clock value.)

`ArgumentLength`

The length, in bytes, of the buffer that the *Argument* parameter points to. This parameter is zero if a buffer is not available.

`Argument`

A pointer to a buffer that contains notification-specific arguments. This parameter is **NULL** if the notification that

the *TransactionNotification* parameter specifies does not require an argument buffer.

For a list of notifications that include an additional argument buffer, see the Remarks section of TRANSACTION_NOTIFICATION.

Return Value

ResourceManagerNotification callback routine must return STATUS_SUCCESS or another status value for which NT_SUCCESS(**status**) equals **TRUE** if the operation is successful.

Typically, the resource manager returns STATUS_SUCCESS if it services the notification synchronously and STATUS_PENDING if it services the notification asynchronously, but it can return STATUS_PENDING in either case. However, the resource manager can return STATUS_PENDING only for notifications that it responds to by calling one of the **Tm*Xxx*Complete** or **Zw*Xxx*Complete** routines.

In other words, if the resource manager must respond to a notification by calling one of the **Tm*Xxx*Complete** or **Zw*Xxx*Complete** routines, it can return STATUS_PENDING for the notification and later call the appropriate **Tm*Xxx*Complete** or **Zw*Xxx*Complete** routine.

If an error occurs, the callback routine must return a status value for which NT_SUCCESS(status) equals **FALSE**.

Remarks

To register a *ResourceManagerNotification* callback routine, your resource manager must call TmEnableCallbacks.

Note that *ResourceManagerNotification* callback routines receive a pointer, instead of a handle, to an enlistment object. You can pass the enlistment object pointer to the enlistment object's TmXxx routines.

For more information about *ResourceManagerNotification* callback routines, see Creating a Resource Manager.

The *ResourceManagerNotification* callback routine is called at IRQL = PASSIVE_LEVEL, and it must return at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	PASSIVE_LEVEL (See Remarks section)

See Also

TRANSACTION_NOTIFICATION

TmEnableCallbacks

ZwCreateEnlistment

ZwGetNotificationResourceManager

PushEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **PushEntryList** routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
void PushEntryList(
  PSINGLE_LIST_ENTRY                  ListHead,
  __drv_aliasesMem PSINGLE_LIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the SINGLE_LIST_ENTRY structure that serves as the list header.

`Entry`

Pointer to SINGLE_LIST_ENTRY structure that represents the entry to be inserted on the list.

Return Value

None

Remarks

PushEntryList sets *ListHead*->**Next** to *Entry*, and *Entry*->**Next** to point to the old first entry of the list.

For information about using this routine when implementing a singly linked list, see Singly and Doubly Linked Lists.

Callers of **PushEntryList** can be running at any IRQL. If **PushEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

ExInterlockedPushEntryList

PopEntryList

READ_PORT_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_UCHAR** routine reads a number of bytes from the specified port address into a buffer.

Syntax

```
NTHALAPI VOID READ_PORT_BUFFER_UCHAR(
  PUCHAR Port,
  PUCHAR Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of UCHAR values is read.

`Count`

Specifies the number of bytes to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **READ_PORT_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_ULONG** routine reads a number of ULONG values from the specified port address into a buffer.

Syntax

```
NTHALAPI VOID READ_PORT_BUFFER_ULONG(
  PULONG Port,
  PULONG Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of ULONG values is read.

`Count`

Specifies the number of ULONG values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **READ_PORT_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_BUFFER_USHORT** routine reads a number of USHORT values from the specified port address into a buffer.

Syntax

```
NTHALAPI VOID READ_PORT_BUFFER_USHORT(
  PUSHORT Port,
  PUSHORT Buffer,
  ULONG   Count
);
```

Parameters

`Port`

Specifies the port address, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer into which an array of USHORT values is read.

`Count`

Specifies the number of USHORT values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **READ_PORT_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

READ_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_UCHAR** routine reads a byte from the specified port address.

Syntax

```
NTHALAPI UCHAR READ_PORT_UCHAR(
  PUCHAR Port
);
```

Parameters

`Port`

Return Value

READ_PORT_UCHAR returns the byte that is read from the specified port address.

Remarks

Callers of **READ_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_ULONG** routine reads a ULONG value from the specified port address.

Syntax

```
NTHALAPI ULONG READ_PORT_ULONG(
  PULONG Port
);
```

Parameters

`Port`

Return Value

READ_PORT_ULONG returns the ULONG value that is read from the specified port address.

Remarks

Callers of **READ_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_PORT_USHORT** routine reads a USHORT value from the specified port address.

Syntax

```
NTHALAPI USHORT READ_PORT_USHORT(
  PUSHORT Port
);
```

Parameters

`Port`

Return Value

READ_PORT_USHORT returns the USHORT value that is read from the specified port address.

Remarks

Callers of **READ_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_UCHAR** routine reads a number of bytes from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of UCHAR values is read.

`Count`

Specifies the number of bytes to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **READ_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_ULONG** routine reads a number of ULONG values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of ULONG values is read.

`Count`

Specifies the number of ULONG values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **READ_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_USHORT** routine reads a number of USHORT values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of USHORT values is read.

`Count`

Specifies the number of USHORT values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **READ_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_UCHAR** routine reads a byte from the specified register address.

Syntax

```
UCHAR READ_REGISTER_UCHAR(
  volatile UCHAR *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_UCHAR returns the byte read from the specified register address.

Remarks

Callers of **READ_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_ULONG** routine reads a ULONG value from the specified register address.

Syntax

```
ULONG READ_REGISTER_ULONG(
  volatile ULONG *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_ULONG returns the ULONG value read from the specified register address.

Remarks

Callers of **READ_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_USHORT** routine reads a USHORT value from the specified register address.

Syntax

```
USHORT READ_REGISTER_USHORT(
  volatile USHORT *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_USHORT returns the USHORT value read from the specified register address.

Remarks

Callers of **READ_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

RemoveEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveEntryList** routine removes an entry from a doubly linked list of LIST_ENTRY structures.

Syntax

```
BOOLEAN RemoveEntryList(
  PLIST_ENTRY Entry
);
```

Parameters

`Entry`

Pointer to the LIST_ENTRY structure that represents the entry to be removed.

Return Value

RemoveEntryList returns **TRUE** if, after removal of the designated entry, the list is empty. Otherwise, the routine returns **FALSE** to indicate that the resulting list still contains one or more entries. For information, see Remarks.

> **Note** Before Microsoft Windows Server 2003, **RemoveEntryList** was defined to return VOID, not BOOLEAN.

Remarks

RemoveEntryList removes the entry by setting the **Flink** member of the entry before *Entry* to point to the entry after *Entry*, and the **Blink** member of the entry after *Entry* to point to the entry before *Entry*.

The return value can be used to detect when the last entry is removed from the list. An empty list consists of a list head only and no list entries.

> **Note** Typically, *Entry* points to an entry in a list and not to the list head. However, *Entry* can point to a list head, in which case the routine removes the list head from the list to produce a headless list. When **RemoveEntryList** is used in this way, the return value should typically be ignored. To determine whether a list is empty, use the IsListEmpty routine.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

In Windows XP and Windows 2000, **RemoveEntryList** did not return a value. Starting with Windows Server 2003, **RemoveEntryList** returns a BOOLEAN value.

Callers of **RemoveEntryList** can be running at any IRQL. If **RemoveEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)
DDI compliance rules	DoubleExFreePool

See Also

InitializeListHead

IsListEmpty

RemoveHeadList

RemoveTailList

RemoveHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveHeadList** routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY RemoveHeadList(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that serves as the list header.

Return Value

RemoveHeadList returns a pointer to the entry removed from the list. If the list is empty, **RemoveHeadList** returns *ListHead*.

Remarks

RemoveHeadList removes the first entry from the list by setting *ListHead*->**Flink** to point to the second entry in the list. The routine sets the **Blink** member of the second entry to *ListHead*. In the event the list is empty, this is effectively a no-op.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **RemoveHeadList** can be running at any IRQL. If **RemoveHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

DDI compliance rules	CancelSpinLock, CompleteRequest, DoubleCompletion, IoAllocateFree, IoReuseIrp, IrpProcessingComplete, MarkingInterlockedQueuedIrps, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, RemoveLock, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove, RemoveLockMnSurpriseRemove, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePnp, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite, InvalidReqAccessLocal, DoubleExFreePool, Init_NdisAllocateIoWorkItem

See Also

ExInterlockedRemoveHeadList

InitializeListHead

IsListEmpty

RemoveEntryList

RemoveTailList

RemoveTailList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveTailList** routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY RemoveTailList(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that serves as the list header.

Return Value

RemoveTailList returns a pointer to the entry that was at the tail of the list. If the list is empty, **RemoveTailList** returns *ListHead*.

Remarks

RemoveTailList removes the last entry from the list by setting *ListHead*->**Blink** to point to the second-to-last entry in the list. The routine sets the **Flink** member of the new first entry to *ListHead*. In the event the list is empty, this is effectively a no-op.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertHeadList** can be running at any IRQL. If **InsertHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

InitializeListHead

IsListEmpty

RemoveEntryList

RemoveHeadList

REQUEST_POWER_COMPLETE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *PowerCompletion* callback routine completes the processing of a power IRP.

Syntax

```
REQUEST_POWER_COMPLETE RequestPowerComplete;

void RequestPowerComplete(
  PDEVICE_OBJECT DeviceObject,
  UCHAR MinorFunction,
  POWER_STATE PowerState,
  PVOID Context,
  PIO_STATUS_BLOCK IoStatus
)
{...}
```

Parameters

`DeviceObject`

A pointer to the target DEVICE_OBJECT for the completed power IRP.

`MinorFunction`

Specifies the minor function code in the power IRP. For more information, see the list of supported **IRP_MN_*XXX*** codes in the Remarks section.

`PowerState`

Specifies the device power state or system power state that was passed to the PoRequestPowerIrp routine.

`Context`

A pointer to the context that was passed to **PoRequestPowerIrp**.

`IoStatus`

A pointer to the IO_STATUS_BLOCK structure for the completed IRP.

Return Value

None

Remarks

A driver that sends a power IRP might need to perform additional tasks after all other drivers have completed the IRP. If so, the sending driver should register a *PowerCompletion* callback routine during the call to the PoRequestPowerIrp routine that allocates the IRP.

A driver's *PowerCompletion* callback routine is used only for IRP_MJ_POWER IRPs that have minor IRP codes of IRP_MN_SET_POWER, IRP_MN_QUERY_POWER, and IRP_MN_WAIT_WAKE. For more information, see Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States and Wait/Wake Callback

Routines.

The I/O manager calls the sending driver's *PowerCompletion* routine only after the I/O manager has called all the IoCompletion routines that were set by other drivers as they passed the IRP down the stack. The *PowerCompletion* routine performs any additional tasks that the sender of the IRP requires after all other drivers have completed the IRP. The *PowerCompletion* routine should not free the IRP—the power manager does that.

In Windows 2000 and later versions of Windows, the *PowerCompletion* routine is called at IRQL = PASSIVE_LEVEL or IRQL = DISPATCH_LEVEL. In Windows 98/Me, the *PowerCompletion* routine is always called at IRQL = PASSIVE_LEVEL, and drivers must complete IRPs at IRQL = PASSIVE_LEVEL.

Examples

To define a *PowerCompletion* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *PowerCompletion* callback routine that is named `MyPowerCompletion`, use the REQUEST_POWER_COMPLETE type as shown in this code example:

```
REQUEST_POWER_COMPLETE MyPowerCompletion;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyPowerCompletion(
    PDEVICE_OBJECT DeviceObject,
    UCHAR MinorFunction,
    POWER_STATE PowerState,
    PVOID Context,
    PIO_STATUS_BLOCK IoStatus
    )
  {
      // Function body
  }
```

The REQUEST_POWER_COMPLETE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the REQUEST_POWER_COMPLETE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows 2000 and later versions of Windows.

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL (see Remarks section).

See Also

DEVICE_OBJECT

IO_STATUS_BLOCK

IRP_MJ_POWER

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IRP_MN_WAIT_WAKE

IoCompletion

PoRequestPowerIrp

RTL_QUERY_REGISTRY_ROUTINE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *QueryRoutine* routine provides information about a registry value that was requested in a preceding call to the RtlQueryRegistryValues routine.

Syntax

```
RTL_QUERY_REGISTRY_ROUTINE RtlQueryRegistryRoutine;

NTSTATUS RtlQueryRegistryRoutine(
  PWSTR ValueName,
  ULONG ValueType,
  PVOID ValueData,
  ULONG ValueLength,
  PVOID Context,
  PVOID EntryContext
)
{...}
```

Parameters

`ValueName`

Specifies the registry key that is associated with the requested registry value. This parameter is a pointer to a null-terminated, Unicode string that contains the key.

`ValueType`

Specifies the type of registry value that is stored with the specified registry key. For more information registry value types, see the definition of the *Type* parameter in KEY_VALUE_BASIC_INFORMATION.

`ValueData`

A pointer to the data value that is associated with the specified registry key. The driver must treat this value as read-only. For more information about the type of value data that *ValueData* points to, see the definition of the *Type* parameter in KEY_VALUE_BASIC_INFORMATION.

`ValueLength`

Specifies the length, in bytes, of the value that *ValueData* points to.

`Context`

Specifies the *Context* parameter value that the driver specified in the preceding call to **RtlQueryRegistryValues**.

`EntryContext`

Specifies an **EntryContext** value in a *QueryTable* array element that the driver specified in the preceding call to **RtlQueryRegistryValues**.

Return Value

QueryRoutine returns STATUS_SUCCESS if the call is successful. Otherwise, it returns an appropriate error status code. Use only status codes that are defined in the Ntstatus.h header file.

Remarks

A kernel-mode driver implements a *QueryRoutine* routine. This routine is called by the RtlQueryRegistryValues routine.

To obtain information about one or more registry values, the driver calls **RtlQueryRegistryValues** and passes a pointer, as an input parameter, to an array of **RTL_QUERY_REGISTRY_TABLE** structures. Each structure in this array contains a pointer to a driver-implemented *QueryRoutine* routine and a request for information about a particular registry value. For each structure in the array, **RtlQueryRegistryValues** calls the specified *QueryRoutine* routine and passes to this routine a set of parameters that contain the requested information about the specified registry value.

For more information about the **RTL_QUERY_REGISTRY_TABLE** structure, see RtlQueryRegistryValues.

Examples

To define a *QueryRoutine* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *QueryRoutine* callback routine that is named `MyQueryRoutine`, use the RTL_QUERY_REGISTRY_ROUTINE type as shown in this code example:

```
RTL_QUERY_REGISTRY_ROUTINE MyQueryRoutine;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyQueryRoutine(
    PWSTR ValueName,
    ULONG ValueType,
    PVOID ValueData,
    ULONG ValueLength,
    PVOID Context,
    PVOID EntryContext
    )
  {
      // Function body
  }
```

The RTL_QUERY_REGISTRY_ROUTINE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the RTL_QUERY_REGISTRY_ROUTINE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

KEY_VALUE_BASIC_INFORMATION

RtlQueryRegistryValues

RtlAnsiStringToUnicodeSize macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAnsiStringToUnicodeSize** routine returns the number of bytes required to hold an ANSI string converted into a Unicode string.

Syntax

```
void RtlAnsiStringToUnicodeSize(
   STRING
);
```

Parameters

`STRING`

Pointer to a buffer containing the ANSI string.

Return Value

RtlAnsiStringToUnicodeSize returns the necessary size in bytes for a Unicode string buffer.

Remarks

> Casting the return value of the **RtlAnsiStringToUnicodeSize** routine to USHORT might cause a loss of data. To avoid potential problems, callers of this routine should test the return value for arithmetic overflow. Note that **RtlAnsiStringToUnicodeSize** accounts for the trailing **NULL**.

Callers of **RtlAnsiStringToUnicodeSize** must be running at IRQL = PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAnsiStringToUnicodeString

RtlxAnsiStringToUnicodeSize

RtlAnsiStringToUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

RtlAnsiStringToUnicodeString converts the given ANSI source string into a Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlAnsiStringToUnicodeString(
  PUNICODE_STRING DestinationString,
  PCANSI_STRING   SourceString,
  BOOLEAN         AllocateDestinationString
);
```

Parameters

`DestinationString`

Pointer to a UNICODE_STRING structure to hold the converted Unicode string. If *AllocateDestinationString* is **TRUE**, the routine allocates a new buffer to hold the string data, and updates the **Buffer** member of *DestinationString* to point to the new buffer. Otherwise, the routine uses the currently-specified buffer to hold the string.

`SourceString`

Pointer to the ANSI string to be converted to Unicode.

`AllocateDestinationString`

Specifies if this routine should allocate the buffer space for the destination string. If it does, the caller must deallocate the buffer by calling RtlFreeUnicodeString.

Return Value

If the conversion succeeds, **RtlAnsiStringToUnicodeString** returns STATUS_SUCCESS. On failure, the routine does not allocate any memory.

Remarks

The translation conforms to the current system locale information.

If caller sets *AllocateDestinationString* to **TRUE**, the routine replaces the **Buffer** member of *DestinationString* with a pointer to the buffer it allocates. The old value can be overwritten even when the routine returns an error status code.

This routine is not declared in a header file. However, you can copy the following declaration to your source code:

```
```

```
NTSYSAPI
WCHAR
NTAPI
RtlAnsiCharToUnicodeChar(
    __inout PUCHAR *SourceCharacter
    );
```

You can use the following routines to convert single-byte and double-byte characters to Unicode characters:

RtlAnsiStringToUnicodeSize

RtlAnsiStringToUnicodeString

RtlMultiByteToUnicodeSize

RtlMultiByteToUnicodeN

For more information about these and other string-handling routines, see Strings.

RtlAnsiCharToUnicodeChar has security deficiencies. Consider using the **RtlAnsiStringToUnicodeString** routine instead.

If, on entry, **SourceCharacter* points to a one-byte buffer that contains the first byte of a two-byte character code, **RtlAnsiCharToUnicodeChar** overruns the buffer when it tries to read the second byte.

At system startup, the operating system determines the user language from the locale settings and installs the appropriate system ANSI code page. **RtlAnsiCharToUnicodeChar** uses this code page to convert a single-byte or double-byte character to the corresponding Unicode character.

If the system ANSI code page defines a single-byte character set (that is, the ANSI character set), **RtlAnsiCharToUnicodeChar** speeds up the conversion operation by simply zero-extending an ANSI character in the range 0x00 to 0x7f to produce the corresponding Unicode character. The routine converts the ANSI value 0x5c to the backslash character ("\"), even if the single-byte code page defines this character as the yen sign.

If, on entry, **SourceCharacter* points to an invalid character code, **RtlAnsiCharToUnicodeChar** returns the Unicode space character code, 0x0020. The following list shows examples of invalid character codes:

- The first byte of the character code is a value that is valid only as the second byte of a two-byte character code.
- The second byte of a two-byte character code is a value that is valid only as the first byte.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ANSI_STRING

RtlAnsiStringToUnicodeSize

RtlFreeUnicodeString

RtlInitAnsiString

RtlUnicodeStringToAnsiString

UNICODE_STRING

RtlAppendUnicodeStringToString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAppendUnicodeStringToString** routine concatenates two Unicode strings.

Syntax

```
NTSYSAPI NTSTATUS RtlAppendUnicodeStringToString(
  PUNICODE_STRING  Destination,
  PCUNICODE_STRING Source
);
```

Parameters

`Destination`

Pointer to a buffered Unicode string.

`Source`

Pointer to the buffered string to be concatenated.

Return Value

RtlAppendUnicodeStringToString can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The source string was successfully appended to the destination counted string. The destination string length is updated to include the appended bytes.
STATUS_BUFFER_TOO_SMALL	The destination string length is too small to allow the source string to be concatenated. Accordingly, the destination string length is not updated.

Remarks

RtlAppendUnicodeStringToString copies bytes from the source to the specified destination string, after the current contents of the destination string and the destination length is incremented by the number of bytes copied.

The *Destination* and *Source* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

RtlAppendUnicodeToString

RtlAppendUnicodeToString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAppendUnicodeToString** routine concatenates the supplied Unicode string to a buffered Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlAppendUnicodeToString(
  PUNICODE_STRING Destination,
  PCWSTR          Source
);
```

Parameters

`Destination`

A pointer to the buffered string.

`Source`

A pointer to the null-terminated string to append to the *Destination* string.

Return Value

RtlAppendUnicodeToString can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The source string was successfully appended to the destination string, and the destination string length was updated to include the appended bytes.
STATUS_BUFFER_TOO_SMALL	The destination string buffer is too small to allow the source string to be appended, so the destination string length is not updated and the contents of the destination string buffer are not changed.

Remarks

RtlAppendUnicodeToString appends the source string to the end of the destination string.

If the destination buffer is not large enough to contain the string that would result from concatenating the source and destination strings (not including the null character that terminates the source string), the routine returns STATUS_BUFFER_TOO_SMALL and leaves the destination string unchanged.

The *Destination* buffer must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	See Remarks section.

See Also

RtlAppendUnicodeStringToString

RtlAreBitsClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAreBitsClear** routine determines whether a given range of bits within a bitmap variable is clear.

Syntax

```
NTSYSAPI BOOLEAN RtlAreBitsClear(
  PRTL_BITMAP BitMapHeader,
  ULONG       StartingIndex,
  ULONG       Length
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

Specifies the start of the bit range to be tested. This is a zero-based value indicating the position of the first bit in the range.

`Length`

Specifies how many bits to test.

Return Value

RtlAreBitsClear returns **TRUE** if *Length*consecutive bits beginning at *StartingIndex* are clear (that is, all the bits from *StartingIndex*to (*StartingIndex*+ *Length*) -1). It returns **FALSE** if any bit in the given range is set, if the given range is not a proper subset of the bitmap, or if the given *Length* is zero.

Remarks

Callers of **RtlAreBitsClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlAreBitsClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsSet

RtlCheckBit

RtlClearAllBits

RtlFindClearBits

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlAreBitsSet function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAreBitsSet** routine determines whether a given range of bits within a bitmap variable is set.

Syntax

```
NTSYSAPI BOOLEAN RtlAreBitsSet(
  PRTL_BITMAP BitMapHeader,
  ULONG       StartingIndex,
  ULONG       Length
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

Specifies the start of the bit range to be tested. This is a zero-based value indicating the position of the first bit in the range.

`Length`

Specifies how many bits to test.

Return Value

RtlAreBitsSet returns **TRUE** if *Length*consecutive bits beginning at *StartingIndex* are set (that is, all the bits from *StartingIndex*to (*StartingIndex*+ *Length*)). It returns **FALSE** if any bit in the given range is clear, if the given range is not a proper subset of the bitmap, or if the given *Length* is zero.

Remarks

Callers of **RtlAreBitsSet** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlAreBitsSet** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlCheckBit

RtlFindSetBits

RtlInitializeBitMap

RtlSetAllBits

RtlCheckBit macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCheckBit** routine determines whether a particular bit in a given bitmap variable is clear or set.

Syntax

```
void RtlCheckBit(
   BMH,
   BP
);
```

Parameters

`BMH`

[in]

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`BP`

[in]

Specifies which bit to check. This is a zero-based value indicating the position of the bit to be tested.

Return Value

RtlCheckBit returns zero if the given bit is clear, or one if the given bit is set.

Remarks

Callers of **RtlCheckBit** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlCheckBit** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	"<= APC_LEVEL (see Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlAreBitsSet

RtlInitializeBitMap

RtlNumberOfClearBits

RtlNumberOfSetBits

RtlCheckRegistryKey function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCheckRegistryKey** routine checks for the existence of a given named key in the registry.

Syntax

```
NTSYSAPI NTSTATUS RtlCheckRegistryKey(
  ULONG RelativeTo,
  PWSTR Path
);
```

Parameters

`RelativeTo`

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following.

VALUE	MEANING
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_USER	Path is relative to the registry settings for the current user. (For a system process, this is **\Users.Default**.) This is equivalent to HKEY_CURRENT_USER in user mode.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_HANDLE	This value should not be passed into this routine. Despite the redundancy of a check for the existence of an already-opened key, it has the side effect of closing the passed handle.

`Path`

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly.

Return Value

If the given named key exists in the registry along the given relative path, **RtlCheckRegistryKey** returns STATUS_SUCCESS.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlQueryRegistryValues

RtlClearAllBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlClearAllBits** routine sets all bits in a given bitmap variable to zero.

Syntax

```
NTSYSAPI VOID RtlClearAllBits(
  PRTL_BITMAP BitMapHeader
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

Return Value

None

Remarks

Callers of **RtlClearAllBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlClearAllBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlAreBitsSet

RtlClearBits

RtlFindSetBits

RtlFindSetBitsAndClear

RtlInitializeBitMap

RtlNumberOfSetBits

RtlClearBit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlClearBit** routine sets the specified bit in a bitmap to zero.

Syntax

```
NTSYSAPI VOID RtlClearBit(
  PRTL_BITMAP BitMapHeader,
  ULONG       BitNumber
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`BitNumber`

Specifies the zero-based index of the bit within the bitmap. The routine sets this bit to zero.

Return Value

None

Remarks

Callers of **RtlClearBit** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlClearBit** can be called at any IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlInitializeBitMap

RtlClearBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlClearBits** routine sets all bits in the specified range of bits in the bitmap to zero.

Syntax

```
NTSYSAPI VOID RtlClearBits(
  PRTL_BITMAP BitMapHeader,
  ULONG       StartingIndex,
  ULONG       NumberToClear
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

The index of the first bit in the bit range that is to be cleared. If the bitmap contains N bits, the bits are numbered from 0 to N-1.

`NumberToClear`

Specifies how many bits to clear. If the bitmap contains N bits, this parameter can be a value in the range 1 to (N - *StartingIndex*).

Return Value

This routine does not return a value.

Remarks

If the *NumberToClear* parameter is zero, **RtlClearBits** simply returns control without clearing any bits.

The sum (*StartingIndex* + *NumberToClear*) must not exceed the *SizeOfBitMap* parameter value specified in the RtlInitializeBitMap call that initialized the bitmap.

Callers of **RtlClearBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlClearBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section.)"

See Also

RTL_BITMAP

RtlAreBitsSet

RtlClearAllBits

RtlFindSetBits

RtlFindSetBitsAndClear

RtlInitializeBitMap

RtlNumberOfSetBits

RtlCmDecodeMemIoResource function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCmDecodeMemIoResource** routine provides the starting address and length of a CM_PARTIAL_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.

Syntax

```
NTSYSAPI ULONGLONG RtlCmDecodeMemIoResource(
  PCM_PARTIAL_RESOURCE_DESCRIPTOR Descriptor,
  PULONGLONG                      Start
);
```

Parameters

`Descriptor`

A pointer to the CM_PARTIAL_RESOURCE_DESCRIPTOR structure to provide the starting address and length for.

`Start`

A pointer to a variable that receives the starting address of the range of memory or I/O port addresses.

Return Value

RtlCmDecodeMemIoResource returns the length of the address range, in bytes.

Remarks

The **Type** member of the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure must be **CmResourceTypeMemory**, **CmResourceTypeMemoryLarge**, or **CmResourceTypePort**.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

RtlCmEncodeMemIoResource

RtlCmEncodeMemIoResource function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCmEncodeMemIoResource** routine updates a CM_PARTIAL_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.

Syntax

```
NTSYSAPI NTSTATUS RtlCmEncodeMemIoResource(
  PCM_PARTIAL_RESOURCE_DESCRIPTOR Descriptor,
  UCHAR                           Type,
  ULONGLONG                       Length,
  ULONGLONG                       Start
);
```

Parameters

`Descriptor`

A pointer to the CM_PARTIAL_RESOURCE_DESCRIPTOR structure to update.

`Type`

The resource type of the memory. This parameter can be **CmResourceTypeMemory**, **CmResourceTypeMemoryLarge**, or **CmResourceTypePort**.

`Length`

The length, in bytes, of the range of allocated addresses.

`Start`

The starting address of the range of memory or I/O port addresses.

Return Value

RtlCmEncodeMemIoResource returns an NTSTATUS value. This routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure has been updated.

STATUS_UNSUCCESSFUL	The specified value for *Length* cannot be encoded in a **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure.
STATUS_INVALID_PARAMETER	One or more of the specified parameters are invalid.

Remarks

Addresses that are larger than 32 bits in length must satisfy certain alignment restrictions or else the routine returns STATUS_UNSUCCESSFUL.

ADDRESS LENGTH	ALIGNMENT RESTRICTION
40 bits	Lowest 8 bits must be zero.
48 bits	Lowest 16 bits must be zero.
64 bits	Lowest 32 bits must be zero.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR

RtlCmDecodeMemIoResource

RtlCompareMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCompareMemory** routine compares two blocks of memory and returns the number of bytes that match.

Syntax

```
NTSYSAPI SIZE_T RtlCompareMemory(
  const VOID *Source1,
  const VOID *Source2,
  SIZE_T     Length
);
```

Parameters

`Source1`

A pointer to the first block of memory.

`Source2`

A pointer to the second block of memory.

`Length`

The number of bytes to compare.

Return Value

RtlCompareMemory returns the number of bytes in the two blocks that match. If all bytes match up to the specified *Length* value, the *Length* value is returned.

Remarks

The routine starts by comparing the first byte in the first block to the first byte in the second block, and continues to compare successive bytes in the two blocks while the bytes match. The routine stops comparing bytes when it encounters the first pair of bytes that are not equal, or when the number of matching bytes equals the *Length* parameter value, whichever occurs first.

Callers of **RtlCompareMemory** can be running at any IRQL if both blocks of memory are resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib; OneCoreUAP.lib on Windows 10
DLL	NtDll.dll (user mode); Kernel32.dll (user mode); NtosKrnl.exe (kernel mode)
IRQL	Any level (See Remarks section)
DDI compliance rules	BufAfterReqCompletedIntIoctlA, BufAfterReqCompletedIoctlA, BufAfterReqCompletedReadA, BufAfterReqCompletedWriteA

RtlCompareUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCompareUnicodeString** routine compares two Unicode strings.

Syntax

```
NTSYSAPI LONG RtlCompareUnicodeString(
  PCUNICODE_STRING String1,
  PCUNICODE_STRING String2,
  BOOLEAN          CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first string.

`String2`

Pointer to the second string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlCompareUnicodeString returns a signed value that gives the results of the comparison:

RETURN CODE	DESCRIPTION
Zero	*String1* equals *String2*.
< Zero	*String1* is less than *String2*.
> Zero	*String1* is greater than *String2*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCompareString

RtlEqualString

RtlConvertLongToLargeInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlConvertLongToLargeInteger** routine converts the input signed integer to a signed large integer.

Syntax

```
DECLSPEC_DEPRECATED_DDK_WINXP LARGE_INTEGER NTAPI_INLINE RtlConvertLongToLargeInteger(
  LONG SignedInteger
);
```

Parameters

`SignedInteger`

Specifies an integer of type LONG.

Return Value

RtlConvertLongToLargeInteger returns the large integer result.

Remarks

This routine is not supported in Windows XP. Use native support for **__int64** instead.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

RtlConvertUlongToLargeInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlConvertUlongToLargeInteger** routine converts the input unsigned integer to a signed large integer. For Windows XP and later versions of Windows, do not use this routine; use the native support for **__int64**.

Syntax

```
DECLSPEC_DEPRECATED_DDK_WINXP LARGE_INTEGER NTAPI_INLINE RtlConvertUlongToLargeInteger(
  ULONG UnsignedInteger
);
```

Parameters

`UnsignedInteger`

Specifies a value of type ULONG.

Return Value

RtlConvertUlongToLargeInteger returns the converted large integer.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

RtlCopyMemory macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCopyMemory** routine copies the contents of a source memory block to a destination memory block.

Syntax

```
void RtlCopyMemory(
   Destination,
   Source,
   Length
);
```

Parameters

`Destination`

Datatype: void*. A pointer to the destination memory block to copy the bytes to.

`Source`

Datatype: const void*. A pointer to the source memory block to copy the bytes from.

`Length`

Datatype: size_t. The number of bytes to copy from the source to the destination.

Return Value

None

Remarks

RtlCopyMemory runs faster than **RtlMoveMemory**. However, **RtlCopyMemory** requires that the source memory block, which is defined by *Source* and *Length*, cannot overlap the destination memory block, which is defined by *Destination* and *Length*. In contrast, **RtlMoveMemory** correctly handles the case in which the source and destination memory blocks overlap.

New drivers should use the **RtlCopyMemory** routine instead of **RtlCopyBytes**.

Callers of **RtlCopyMemory** can be running at any IRQL if the source and destination memory blocks are in nonpaged system memory. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtDll.dll (user mode); NtosKrnl.exe (kernel mode)
IRQL	Any level (See Remarks section)

See Also

RtlMoveMemory

RtlCopyUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCopyUnicodeString** routine copies a source string to a destination string.

Syntax

```
NTSYSAPI VOID RtlCopyUnicodeString(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString
);
```

Parameters

`DestinationString`

A pointer to the destination string buffer. This parameter points to a UNICODE_STRING structure.

`SourceString`

A pointer to the source string buffer. This parameter points to a **UNICODE_STRING** structure.

Return Value

None

Remarks

If *SourceString* is **NULL**, this routine sets the **Length** field of the structure pointed to by *DestinationString* to zero.

This routine does not modify the **MaximumLength** and **Buffer** fields of the structure pointed to by *DestinationString*.

The number of bytes copied from the source string is either the source string length (specified by the **Length** member of the structure pointed to by *SourceString*) or the maximum length of the destination string (specified by the **MaximumLength** member of the structure pointed to by *DestinationString*), whichever is smaller.

The caller must properly initialize all members of the structure pointed to by *DestinationString* before calling **RtlCopyUnicodeString**. Failure to initialize the **Length** or the **MaximumLength** member before calling this routine can cause a buffer overrun.

The *DestinationString* and *SourceString* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	Any level (See Remarks section)

See Also

RtlCopyString

UNICODE_STRING

RtlCreateRegistryKey function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCreateRegistryKey** routine adds a key object in the registry along a given relative path.

Syntax

```
NTSYSAPI NTSTATUS RtlCreateRegistryKey(
  ULONG RelativeTo,
  PWSTR Path
);
```

Parameters

`RelativeTo`

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following.

VALUE	MEANING
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**. (For a system process, this is **\Users.Default**.)

`Path`

Specifies the registry path according to the *RelativeTo* value.

Return Value

RtlCreateRegistryKey returns STATUS_SUCCESS if the key is created.

Remarks

RtlCreateRegistryKey creates only the keys that you specify. It does not create missing keys in the path to the specified key. If any keys in the path are missing, the call will fail. To ensure that all keys in the path exist, you must call **RtlCreateRegistryKey** once for each key in the path.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCheckRegistryKey

RtlDeleteRegistryValue

RtlQueryRegistryValues

RtlWriteRegistryValue

ZwEnumerateKey

ZwOpenKey

RtlCreateSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCreateSecurityDescriptor** routine initializes a new absolute-format security descriptor. On return, the security descriptor is initialized with no system ACL, no discretionary ACL, no owner, no primary group, and all control flags set to zero.

Syntax

```
NTSYSAPI NTSTATUS RtlCreateSecurityDescriptor(
  PSECURITY_DESCRIPTOR SecurityDescriptor,
  ULONG                Revision
);
```

Parameters

`SecurityDescriptor`

Pointer to the buffer for the SECURITY_DESCRIPTOR to be initialized.

`Revision`

Specifies the revision level to assign to the security descriptor. Set this parameter to SECURITY_DESCRIPTOR_REVISION.

Return Value

RtlCreateSecurityDescriptor can return one of the following.

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The call completed successfully.
STA TUS _UN KN OW N_R EVIS ION	The caller specified an unsupported value for *Revision*.

Remarks

A successful call to this routine initializes a security descriptor. The fields in this descriptor are set to initial values that indicate that there are no security constraints.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlLengthSecurityDescriptor

RtlSetDaclSecurityDescriptor

RtlValidSecurityDescriptor

SECURITY_DESCRIPTOR

RtlDeleteRegistryValue function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlDeleteRegistryValue** routine removes the specified entry name and the associated values from the registry along the given relative path.

Syntax

```
NTSYSAPI NTSTATUS RtlDeleteRegistryValue(
  ULONG  RelativeTo,
  PCWSTR Path,
  PCWSTR ValueName
);
```

Parameters

`RelativeTo`

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following.

VALUE	MEANING
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**. (For a system process, this is **\Users.Default**.)
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

`Path`

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly.

`ValueName`

Pointer to the value name to be removed from the registry.

Return Value

RtlDeleteRegistryValue returns STATUS_SUCCESS if the value entry was deleted.

Note that if *RelativeTo* is set to RTL_REGISTRY_HANDLE, the following occurs:

- On Windows 98/Me and Windows NT 4.0, the routine closes the specified handle before returning.
- On Windows 2000 and later versions of Windows, the routine leaves the handle open.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlRtlPassive, HwStorPortProhibitedDDIs

See Also

RtlCheckRegistryKey

RtlQueryRegistryValues

RtlWriteRegistryValue

ZwEnumerateKey

ZwOpenKey

RtlDowncaseUnicodeChar function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlDowncaseUnicodeChar** routine converts the specified Unicode character to lowercase.

Syntax

```
NTSYSAPI WCHAR RtlDowncaseUnicodeChar(
  WCHAR SourceCharacter
);
```

Parameters

`SourceCharacter`

Specifies the character to convert.

Return Value

RtlDowncaseUnicodeChar returns the lowercase value of the Unicode character.

Remarks

Ntoskrnl.lib supports the **RtlDowncaseUnicodeChar** routine in the WDK for Windows 7 and later versions of Windows. Ntdll.dll exports the **RtlDowncaseUnicodeChar** entry point in Windows XP and later versions of Windows.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Ntoskrnl.lib; Ntdll.lib
DLL	Ntoskrnl.exe; Ntdll.dll
IRQL	PASSIVE_LEVEL

See Also

RtlDowncaseUnicodeString

RtlUpcaseUnicodeChar

RtlEqualMemory macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlEqualMemory** routine compares two blocks of memory to determine whether the specified number of bytes are identical.

Syntax

```
void RtlEqualMemory(
   Destination,
   Source,
   Length
);
```

Parameters

`Destination`

Datatype: void*. A pointer to a caller-allocated block of memory to compare.

`Source`

Datatype: void*. A pointer to a caller-allocated block of memory that is compared to the block of memory to which *Source1* points.

`Length`

Datatype: size_t. Specifies the number of bytes to be compared.

Return Value

RtlEqualMemory returns TRUE if Source1 and Source2 are equivalent; otherwise, it returns FALSE.

Remarks

RtlEqualMemory begins the comparison with byte zero of each block.

Callers of **RtlEqualMemory** can be running at any IRQL if both blocks of memory are resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level (See Remarks section)

See Also

RtlCompareMemory

RtlEqualUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlEqualUnicodeString** routine compares two Unicode strings to determine whether they are equal.

Syntax

```
NTSYSAPI BOOLEAN RtlEqualUnicodeString(
  PCUNICODE_STRING String1,
  PCUNICODE_STRING String2,
  BOOLEAN          CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first Unicode string.

`String2`

Pointer to the second Unicode string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlEqualUnicodeString returns **TRUE** if the two Unicode strings are equal; otherwise, it returns **FALSE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlEqualString

RtlFillMemory macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFillMemory** routine fills a block of memory with the specified fill value.

Syntax

```
void RtlFillMemory(
   Destination,
   Length,
   Fill
);
```

Parameters

`Destination`

Datatype: void*. A pointer to the block of memory to be filled.

`Length`

Datatype: size_t. The number of bytes in the block of memory to be filled.

`Fill`

Datatype: int. The value to fill the destination memory block with. This value is copied to every byte in the memory block that is defined by *Destination* and *Length*.

Return Value

None

Remarks

Callers of **RtlFillMemory** can be running at any IRQL if the destination memory block is in nonpaged system memory. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtDll.lib (user mode); NtosKrnl.lib (kernel mode)
DLL	NtDll.dll (user mode); NtosKrnl.exe (kernel mode)

IRQL	Any level (See Remarks section)

See Also

RtlZeroMemory

RtlFindClearBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindClearBits** routine searches for a range of clear bits of a requested size within a bitmap.

Syntax

```
NTSYSAPI ULONG RtlFindClearBits(
  PRTL_BITMAP BitMapHeader,
  ULONG       NumberToFind,
  ULONG       HintIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`NumberToFind`

Specifies how many contiguous clear bits will satisfy this request.

`HintIndex`

Specifies a zero-based bit position from which to start looking for a clear bit range of the given size.

Return Value

RtlFindClearBits either returns the zero-based starting bit index for a clear bit range of at least the requested size, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap.

Remarks

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindClearBits** searches the whole bitmap to locate a clear bit range of the requested size. However, it starts searching for the requested range from *HintIndex*, so callers can find such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindClearBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindClearBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBitsAndSet

RtlFindFirstRunClear

RtlFindLongestRunClear

RtlFindSetBits

RtlInitializeBitMap

RtlNumberOfClearBits

RtlFindClearBitsAndSet function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindClearBitsAndSet** routine searches for a range of clear bits of a requested size within a bitmap and sets all bits in the range when it has been located.

Syntax

```
NTSYSAPI ULONG RtlFindClearBitsAndSet(
  PRTL_BITMAP BitMapHeader,
  ULONG       NumberToFind,
  ULONG       HintIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`NumberToFind`

Specifies how many contiguous clear bits will satisfy this request.

`HintIndex`

Specifies a zero-based bit position from which to start looking for a clear bit range of the given size.

Return Value

RtlFindClearBitsAndSet either returns the zero-based starting bit index for a clear bit range of the requested size that it set, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Remarks

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindClearBitsAndSet** searches the whole bitmap to locate a clear bit range of the requested size. However, it starts searching for the requested range from *HintIndex*, so callers can have such a range reset more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindClearBitsAndSet** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindClearBitsAndSet** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfClearBits

RtlSetAllBits

RtlSetBits

RtlFindClearRuns function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindClearRuns** routine finds the specified number of runs of clear bits within a given bitmap.

Syntax

```
NTSYSAPI ULONG RtlFindClearRuns(
  PRTL_BITMAP     BitMapHeader,
  PRTL_BITMAP_RUN RunArray,
  ULONG           SizeOfRunArray,
  BOOLEAN         LocateLongestRuns
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`RunArray`

Pointer to the first element in a caller-allocated array for the bit position and length of each clear run found in the given bitmap variable.

`SizeOfRunArray`

Specifies the maximum number of clear runs to satisfy this request.

`LocateLongestRuns`

If **TRUE**, specifies that the routine is to search the entire bitmap for the longest clear runs it can find. Otherwise, the routine stops searching when it has found the number of clear runs specified by *SizeOfRunArray*.

Return Value

RtlFindClearRuns returns the number of clear runs found.

Remarks

If *LocateLongestRuns*is **TRUE**, the clear runs indicated at *RunArray*are sorted from longest to shortest. A clear run can consist of a single bit.

Callers of **RtlFindClearRuns** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindClearRuns** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlFindFirstRunClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindFirstRunClear** routine searches for the initial contiguous range of clear bits within a given bitmap.

Syntax

```
NTSYSAPI ULONG RtlFindFirstRunClear(
  PRTL_BITMAP BitMapHeader,
  PULONG      StartingIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

Pointer to a variable in which the starting index of the initial clear run in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the returned range. Its value is meaningless if **RtlFindFirstRunClear** cannot find a run of clear bits.

Return Value

RtlFindFirstRunClear returns either the number of bits in the run beginning at *StartingIndex*, or zero if it cannot find a run of clear bits within the bitmap.

Remarks

A returned run can have a single clear bit. That is, once a clear bit is found, **RtlFindFirstRunClear** continues searching until it finds the next set bit, and then returns the number of clear bits in the run it found.

Callers of **RtlFindFirstRunClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindFirstRunClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfClearBits

RtlSetBits

RtlFindLastBackwardRunClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindLastBackwardRunClear** routine searches a given bitmap for the preceding clear run of bits, starting from the specified index position.

Syntax

```
NTSYSAPI ULONG RtlFindLastBackwardRunClear(
  PRTL_BITMAP BitMapHeader,
  ULONG       FromIndex,
  PULONG      StartingRunIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`FromIndex`

Specifies a zero-based bit position at which to start looking for a clear run of bits.

`StartingRunIndex`

Pointer to a variable in which the starting index of the clear run found in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the run preceding the given *FromIndex*. Its value is meaningless if **RtlFindLastBackwardRunClear** cannot find a run of clear bits.

Return Value

RtlFindLastBackwardRunClear returns the number of bits in the run beginning at *StartingRunIndex*, or zero if it cannot find a run of clear bits preceding *FromIndex* in the bitmap.

Remarks

On checked builds of Windows Vista and earlier versions of Windows, the internal implementation of the **RtlFindLastBackwardRunClear** routine can fail a bogus assertion if the caller is running at IRQL = DISPATCH_LEVEL. The failure of this assertion is harmless and can safely be ignored.

Callers of **RtlFindLastBackwardRunClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindLastBackwardRunClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (see Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlFindLeastSignificantBit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindLeastSignificantBit** routine returns the zero-based position of the least significant nonzero bit in its parameter.

Syntax

```
NTSYSAPI CCHAR RtlFindLeastSignificantBit(
  ULONGLONG Set
);
```

Parameters

`Set`

The 64-bit value to be searched for its least significant nonzero bit.

Return Value

The zero-based bit position of the least significant nonzero bit, or -1 if every bit is zero.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

RtlFindMostSignificantBit

RtlFindLongestRunClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindLongestRunClear** routine searches for the largest contiguous range of clear bits within a given bitmap.

Syntax

```
NTSYSAPI ULONG RtlFindLongestRunClear(
  PRTL_BITMAP BitMapHeader,
  PULONG      StartingIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

Pointer to a variable in which the starting index of the longest clear run in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the returned range.

Return Value

RtlFindLongestRunClear returns either the number of bits in the run beginning at *StartingIndex*, or zero if it cannot find a run of clear bits within the bitmap.

Remarks

A returned run can have a single clear bit.

Callers of **RtlFindLongestRunClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindLongestRunClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfClearBits

RtlSetBits

RtlFindMostSignificantBit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindMostSignificantBit** routine returns the zero-based position of the most significant nonzero bit in its parameter.

Syntax

```
NTSYSAPI CCHAR RtlFindMostSignificantBit(
  ULONGLONG Set
);
```

Parameters

`Set`

The 64-bit value to be searched for its most significant nonzero bit.

Return Value

The zero-based bit position of the most significant nonzero bit, or -1 if every bit is zero.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

RtlFindLeastSignificantBit

RtlFindNextForwardRunClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindNextForwardRunClear** routine searches a given bitmap variable for the next clear run of bits, starting from the specified index position.

Syntax

```
NTSYSAPI ULONG RtlFindNextForwardRunClear(
  PRTL_BITMAP BitMapHeader,
  ULONG       FromIndex,
  PULONG      StartingRunIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`FromIndex`

Specifies a zero-based bit position at which to start looking for a clear run of bits.

`StartingRunIndex`

Pointer to a variable in which the starting index of the clear run found in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the run. Its value is meaningless if **RtlFindNextForwardRunClear** cannot find a run of clear bits.

Return Value

RtlFindNextForwardRunClear returns either the number of bits in the run beginning at *StartingRunIndex*, or zero if it cannot find a run of clear bits following *FromIndex* in the bitmap.

Remarks

Callers of **RtlFindNextForwardRunClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindNextForwardRunClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlInitializeBitMap

RtlFindSetBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindSetBits** routine searches for a range of set bits of a requested size within a bitmap.

Syntax

```
NTSYSAPI ULONG RtlFindSetBits(
  PRTL_BITMAP BitMapHeader,
  ULONG       NumberToFind,
  ULONG       HintIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`NumberToFind`

Specifies how many contiguous set bits will satisfy this request.

`HintIndex`

Specifies a zero-based bit position around which to start looking for a set bit range of the given size.

Return Value

RtlFindSetBits either returns the zero-based starting bit index for a set bit range of the requested size, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Remarks

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindSetBits** searches the whole bitmap to locate a set bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can find such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindSetBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindSetBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsSet

RtlClearBits

RtlFindClearBits

RtlFindSetBitsAndClear

RtlInitializeBitMap

RtlNumberOfSetBits

RtlFindSetBitsAndClear function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFindSetBitsAndClear** routine searches for a range of set bits of a requested size within a bitmap and clears all bits in the range when it has been located.

Syntax

```
NTSYSAPI ULONG RtlFindSetBitsAndClear(
  PRTL_BITMAP BitMapHeader,
  ULONG       NumberToFind,
  ULONG       HintIndex
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`NumberToFind`

Specifies how many contiguous set bits will satisfy this request.

`HintIndex`

Specifies a zero-based bit position around which to start looking for a set bit range of the given size.

Return Value

RtlFindSetBitsAndClear either returns the zero-based starting bit index for a set bit range of the requested size that it cleared, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Remarks

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindSetBitsAndClear** searches the whole bitmap to locate a set bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can clear such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindSetBitsAndClear** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlFindSetBitsAndClear** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsSet

RtlClearAllBits

RtlClearBits

RtlFindSetBits

RtlInitializeBitMap

RtlNumberOfSetBits

RtlFreeAnsiString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFreeAnsiString** routine releases storage that was allocated by **RtlUnicodeStringToAnsiString**.

Syntax

```
NTSYSAPI VOID RtlFreeAnsiString(
  PANSI_STRING AnsiString
);
```

Parameters

`AnsiString`

Pointer to the ANSI string buffer previously allocated by **RtlUnicodeStringToAnsiString**.

Return Value

None

Remarks

This routine does not release the Unicode string buffer passed to **RtlUnicodeStringToAnsiString**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

RtlUnicodeStringToAnsiString

RtlFreeUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFreeUnicodeString** routine releases storage that was allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Syntax

```
NTSYSAPI VOID RtlFreeUnicodeString(
  PUNICODE_STRING UnicodeString
);
```

Parameters

`UnicodeString`

Pointer to the Unicode string buffer previously allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Return Value

None

Remarks

This routine does not release the ANSI string buffer passed to **RtlAnsiStringToUnicodeString**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

RtlAnsiStringToUnicodeString

RtlUpcaseUnicodeString

RtlGetVersion function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlGetVersion** routine returns version information about the currently running operating system.

Syntax

```
NTSYSAPI NTSTATUS RtlGetVersion(
  PRTL_OSVERSIONINFOW lpVersionInformation
);
```

Parameters

`lpVersionInformation`

Pointer to either a RTL_OSVERSIONINFOW structure or a RTL_OSVERSIONINFOEXW structure that contains the version information about the currently running operating system. A caller specifies which input structure is used by setting the **dwOSVersionInfoSize** member of the structure to the size in bytes of the structure that is used.

Return Value

RtlGetVersion returns STATUS_SUCCESS.

Remarks

RtlGetVersion is the kernel-mode equivalent of the user-mode **GetVersionEx** function in the Windows SDK. See the example in the Windows SDK that shows how to get the system version.

When using **RtlGetVersion** to determine whether a particular version of the operating system is running, a caller should check for version numbers that are greater than or equal to the required version number. This ensures that a version test succeeds for later versions of Windows.

Because operating system features can be added in a redistributable DLL, checking only the major and minor version numbers is not the most reliable way to verify the presence of a specific system feature. A driver should use RtlVerifyVersionInfo to test for the presence of a specific system feature.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

PsGetVersion

RtlGUIDFromString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlGUIDFromString** routine converts the given Unicode string to a GUID in binary format.

Syntax

```
NTSYSAPI NTSTATUS RtlGUIDFromString(
  PCUNICODE_STRING GuidString,
  GUID             *Guid
);
```

Parameters

`GuidString`

Pointer to the buffered Unicode string to be converted to a GUID.

`Guid`

Pointer to a caller-supplied variable in which the GUID is returned.

Return Value

If the conversion succeeds, **RtlGUIDFromString** returns STATUS_SUCCESS. Otherwise, no conversion was done.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlStringFromGUID

RtlHashUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlHashUnicodeString** routine creates a hash value from a given Unicode string and hash algorithm.

Syntax

```
NTSYSAPI NTSTATUS RtlHashUnicodeString(
  PCUNICODE_STRING String,
  BOOLEAN          CaseInSensitive,
  ULONG            HashAlgorithm,
  PULONG           HashValue
);
```

Parameters

`String`

A pointer to a UNICODE_STRING structure that contains the Unicode string to be converted to a hash value.

`CaseInSensitive`

Specifies whether to treat the Unicode string as case sensitive when computing the hash value. If *CaseInSensitive* is **TRUE**, a lowercase and uppercase string hash to the same value.

`HashAlgorithm`

The hash algorithm to use. If *HashAlgorithm* is HASH_STRING_ALGORITHM_X65599, **RtlHashUnicodeString** uses the x65599 hashing algorithm. If *HashAlgorithm* is HASH_STRING_ALGORITHM_DEFAULT, **RtlHashUnicodeString** uses the default algorithm. Currently, the default algorithm is the x65599 hashing algorithm.

`HashValue`

A pointer to a ULONG variable that receives the hash value.

Return Value

RtlHashUnicodeString returns STATUS_SUCCESS on success, or the appropriate NTSTATUS value on failure. The routine returns a STATUS_INVALID_PARAMETER value if the Unicode string is **NULL**, *HashValue* is **NULL**, or the caller specifies an undefined value for *HashAlgorithm*.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

UNICODE_STRING

RtlInitAnsiString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInitAnsiString** routine initializes a counted string of ANSI characters.

Syntax

```
NTSYSAPI VOID RtlInitAnsiString(
  PANSI_STRING          DestinationString,
  __drv_aliasesMem PCSZ SourceString
);
```

Parameters

`DestinationString`

A pointer to the ANSI_STRING structure to be initialized.

`SourceString`

A pointer to a null-terminated character string. This string is used to initialize the counted string pointed to by *DestinationString*.

Return Value

None

Remarks

This routine initializes a counted character string.

The routine copies the *SourceString* pointer value to the **Buffer** member of the ANSI_STRING structure pointed to by *DestinationString*. The **Length** member of this structure is set to the length, in bytes, of the source string, excluding the terminating null. The **MaximumLength** member of the structure is set to the length, in bytes, of the source string, including the terminating null. If *SourceString* is **NULL**, **Length** and **MaximumLength** are both set to zero.

RtlInitAnsiString does not alter the source string pointed to by *SourceString*.

> **Note** If the source string is longer than MAXUSHORT - 1 bytes, **RtlInitAnsiString** sets the **Length** member of the **ANSI_STRING** structure pointed to by *DestinationString* to MAXUSHORT - 1, and sets the **MaximumLength** member of this structure to MAXUSHORT. In this case, the **Length** and **MaximumLength** values misrepresent the length of the null-terminated source string, and relying on the accuracy of these values is potentially dangerous.

Callers of **RtlInitAnsiString** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL = PASSIVE_LEVEL because most other **Rtl*Xxx*String** routines cannot be called at IRQL > PASSIVE_LEVEL.

The **RTL_CONSTANT_STRING** macro creates a string or Unicode string structure to hold a counted string.

```
STRING RTL_CONSTANT_STRING(
  [in]  PCSZ SourceString
);

UNICODE_STRING RTL_CONSTANT_STRING(
  [in]  PCWSTR SourceString
);
```

RTL_CONSTANT_STRING returns either a string structure or Unicode string structure.

The **RTL_CONSTANT_STRING** macro replaces the **RtlInitAnsiString**, RtlInitString, and RtlInitUnicodeString routines when passing a constant string.

You can use **RTL_CONSTANT_STRING** to initialize global variables.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (See Remarks section)

See Also

ANSI_STRING

RtlInitializeBitMap function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInitializeBitMap** routine initializes the header of a bitmap variable.

Syntax

```
NTSYSAPI VOID RtlInitializeBitMap(
  PRTL_BITMAP             BitMapHeader,
  __drv_aliasesMem PULONG BitMapBuffer,
  ULONG                   SizeOfBitMap
);
```

Parameters

`BitMapHeader`

Pointer to an empty RTL_BITMAP structure.

`BitMapBuffer`

Pointer to caller-allocated memory for the bitmap itself. The base address of this buffer must be ULONG-aligned. The size of the allocated buffer must be an integer multiple of **sizeof**(ULONG) bytes.

`SizeOfBitMap`

Specifies the number of bits in the bitmap. This value can be any number of bits that will fit in the buffer allocated for the bitmap.

Return Value

None

Remarks

A driver can use a bitmap variable as an economical way to keep track of a set of reusable items. For example, file systems use a bitmap variable to track which clusters/sectors on a disk have already been allocated to hold file data. The system-supplied SCSI port driver uses a bitmap variable to track which queue tags have been assigned to SCSI request blocks (SRBs).

RtlInitializeBitMap must be called before any other **Rtl*Xxx*** routine that operates on a bitmap variable. The *BitMapHeader* pointer is an input parameter in all subsequent **Rtl*Xxx*** calls that operate on the caller's bitmap variable at *BitMapBuffer*. The caller is responsible for synchronizing access to the bitmap variable.

RtlInitializeBitMap initializes the caller-supplied **RTL_BITMAP** structure by copying the caller-supplied *BitMapBuffer* and *SizeOfBitMap* values into it. Subsequently, the structure can be passed to other routines to manipulate the bitmap. **RtlInitializeBitMap** does not modify the contents of the bitmap.

Callers of **RtlInitializeBitMap**, and callers of other **Rtl*Xxx*** routines that operate on an initialized bitmap variable, must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitmapHeader* is pageable. Otherwise, **RtlInitializeBitMap** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL (see Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlAreBitsSet

RtlCheckBit

RtlClearAllBits

RtlClearBits

RtlFindClearBits

RtlFindClearBitsAndSet

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlFindSetBits

RtlFindSetBitsAndClear

RtlNumberOfClearBits

RtlNumberOfSetBits

RtlSetAllBits

RtlSetBits

RtlInitString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInitString** routine initializes a counted string of 8-bit characters.

Syntax

```
NTSYSAPI VOID RtlInitString(
  PSTRING              DestinationString,
  __drv_aliasesMem PCSZ SourceString
);
```

Parameters

`DestinationString`

A pointer to the **STRING** structure to be initialized. The Ntdef.h header file defines this structure to be identical to the ANSI_STRING structure.

`SourceString`

A pointer to a null-terminated character string. This string is used to initialize the counted string pointed to by *DestinationString*.

Return Value

None

Remarks

This routine initializes a counted character string.

The routine copies the *SourceString* pointer value to the **Buffer** member of the **STRING** structure pointed to by *DestinationString*. The **Length** member of this structure is set to the length, in bytes, of the source string, excluding the terminating null. The **MaximumLength** member of the structure is set to the length, in bytes, of the source string, including the terminating null. If *SourceString* is **NULL**, **Length** and **MaximumLength** are both set to zero.

RtlInitString does not alter the source string pointed to by *SourceString*.

> **Note** If the source string is longer than MAXUSHORT - 1 bytes, **RtlInitString** sets the **Length** member of the **STRING** structure pointed to by *DestinationString* to MAXUSHORT - 1, and sets the **MaximumLength** member of this structure to MAXUSHORT. In this case, the **Length** and **MaximumLength** values misrepresent the length of the null-terminated source string, and relying on the accuracy of these values is potentially dangerous.

Callers of **RtlInitString** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL = PASSIVE_LEVEL because most other **Rtl*Xxx*String** routines cannot be called at IRQL > PASSIVE_LEVEL.

The **RTL_CONSTANT_STRING** macro creates a string or Unicode string structure to hold a counted string.

```
STRING RTL_CONSTANT_STRING(
  [in]  PCSZ SourceString
);

UNICODE_STRING RTL_CONSTANT_STRING(
  [in]  PCWSTR SourceString
);
```

RTL_CONSTANT_STRING returns either a string structure or Unicode string structure.

The **RTL_CONSTANT_STRING** macro replaces the RtlInitAnsiString, **RtlInitString**, and RtlInitUnicodeString routines when passing a constant string.

You can use **RTL_CONSTANT_STRING** to initialize global variables.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

ANSI_STRING

RtlInitStringEx function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInitStringEx** routine initializes a counted string of 8-bit characters.

Syntax

```
NTSYSAPI NTSTATUS RtlInitStringEx(
  PSTRING              DestinationString,
  __drv_aliasesMem PCSZ SourceString
);
```

Parameters

`DestinationString`

A pointer to the **STRING** structure to be initialized. The Ntdef.h header file defines this structure to be identical to the ANSI_STRING structure.

`SourceString`

A pointer to a null-terminated character string. This string is used to initialize the counted string pointed to by *DestinationString*.

Return Value

Returns STATUS_NAME_TOO_LONG if the SourceString is too long. Otherwise, this routine returns STATUS_SUCCESS.

Remarks

This routine initializes a counted character string.

The routine copies the *SourceString* pointer value to the **Buffer** member of the **STRING** structure pointed to by *DestinationString*. The **Length** member of this structure is set to the length, in bytes, of the source string, excluding the terminating null. The **MaximumLength** member of the structure is set to the length, in bytes, of the source string, including the terminating null. If *SourceString* is **NULL**, **Length** and **MaximumLength** are both set to zero.

RtlInitStringEx does not alter the source string pointed to by *SourceString*.

Callers of **RtlInitStringEx** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL = PASSIVE_LEVEL because most other **Rtl*Xxx*String** routines cannot be called at IRQL > PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 10.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

ANSI_STRING

RtlInitUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Syntax

```
NTSYSAPI VOID RtlInitUnicodeString(
  PUNICODE_STRING         DestinationString,
  __drv_aliasesMem PCWSTR SourceString
);
```

Parameters

`DestinationString`

For more information, see the WdmlibRtlInitUnicodeStringEx function.

`SourceString`

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Return Value

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Remarks

The **RTL_CONSTANT_STRING** macro creates a string or Unicode string structure to hold a counted string.

```
STRING RTL_CONSTANT_STRING(
  [in]  PCSZ SourceString
);

UNICODE_STRING RTL_CONSTANT_STRING(
  [in]  PCWSTR SourceString
);
```

RTL_CONSTANT_STRING returns either a string structure or Unicode string structure.

The **RTL_CONSTANT_STRING** macro replaces the RtlInitAnsiString, RtlInitString, and **RtlInitUnicodeString** routines when passing a constant string.

You can use **RTL_CONSTANT_STRING** to initialize global variables.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	For more information, see the Remarks section of the WdmlibRtlInitUnicodeStringEx function.

See Also

RtlUnicodeStringInit

RtlUnicodeStringInitEx

UNICODE_STRING

WdmlibRtlInitUnicodeStringEx

RtlInt64ToUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInt64ToUnicodeString** routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.

Syntax

```
NTSYSAPI NTSTATUS RtlInt64ToUnicodeString(
  ULONGLONG       Value,
  ULONG           Base,
  PUNICODE_STRING String
);
```

Parameters

`Value`

Specifies the ULONGLONG value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to the UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlInt64ToUnicodeString returns an NTSTATUS value. Possible return values include :

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The routine successfully converted *Value* to a Unicode string.

STA TUS _BU FFE R_O VER FLO W	*Value* is too large to convert, or the **UNICODE_STRING** structure is too small to hold the result.
STA TUS _IN VAL ID_ PAR AM ETE R	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlIntegerToUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIntegerToUnicodeString** routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.

Syntax

```
NTSYSAPI NTSTATUS RtlIntegerToUnicodeString(
  ULONG           Value,
  ULONG           Base,
  PUNICODE_STRING String
);
```

Parameters

`Value`

Specifies the ULONG value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to a UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlIntegerToUnicodeString returns an NTSTATUS value. Possible return values include:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The routine successfully converted *Value* to a Unicode string.

STA TUS _BU FFE R_O VER FLO W	*Value* is too large to convert, or the **UNICODE_STRING** structure is too small to hold the result.
STA TUS _IN VAL ID_ PAR AM ETE R	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlIntPtrToUnicodeString macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIntPtrToUnicodeString** routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.

Syntax

```
void RtlIntPtrToUnicodeString(
   Value,
   Base,
   String
);
```

Parameters

`Value`

Specifies the ULONG_PTR value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to a UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlIntPtrToUnicodeString returns an NTSTATUS value. Possible return values include :

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully converted Value to a Unicode string.
STATUS_BUFFER_OVERFLOW	Value is too large to convert, or the UNICODE_STRING structure is too small to hold the result.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlIntegerToUnicodeString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlIoDecodeMemIoResource function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIoDecodeMemIoResource** routine provides the address information that is contained in an IO_RESOURCE_DESCRIPTOR structure that describes a range of memory or I/O port addresses.

Syntax

```
NTSYSAPI ULONGLONG RtlIoDecodeMemIoResource(
  PIO_RESOURCE_DESCRIPTOR Descriptor,
  PULONGLONG              Alignment,
  PULONGLONG              MinimumAddress,
  PULONGLONG              MaximumAddress
);
```

Parameters

`Descriptor`

A pointer to the IO_RESOURCE_DESCRIPTOR structure to provide the address information for.

`Alignment`

A pointer to a variable that receives the alignment, in bytes, of the address range. This parameter can be **NULL**.

`MinimumAddress`

A pointer to a variable that receives the minimum address of the address range. This parameter can be **NULL**.

`MaximumAddress`

A pointer to a variable that receives the maximum address of the address range. This parameter can be **NULL**.

Return Value

RtlIoDecodeMemIoResource returns the length of the address range, in bytes.

Remarks

The **Type** member of the IO_RESOURCE_DESCRIPTOR structure must be **CmResourceTypeMemory**, **CmResourceTypeMemoryLarge**, or **CmResourceTypePort**.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IO_RESOURCE_DESCRIPTOR

RtlIoEncodeMemIoResource

RtlIoEncodeMemIoResource function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIoEncodeMemIoResource** routine updates an IO_RESOURCE_DESCRIPTOR structure to describe a range of memory or I/O port addresses.

Syntax

```
NTSYSAPI NTSTATUS RtlIoEncodeMemIoResource(
  PIO_RESOURCE_DESCRIPTOR Descriptor,
  UCHAR                   Type,
  ULONGLONG               Length,
  ULONGLONG               Alignment,
  ULONGLONG               MinimumAddress,
  ULONGLONG               MaximumAddress
);
```

Parameters

`Descriptor`

A pointer to the IO_RESOURCE_DESCRIPTOR structure to update.

`Type`

The resource type of the address range. This parameter can be **CmResourceTypeMemory**, **CmResourceTypeMemoryLarge**, or **CmResourceTypePort**.

`Length`

The length, in bytes, of the range of assignable addresses.

`Alignment`

The alignment, in bytes, of the starting address of address range.

`MinimumAddress`

The minimum address that can be assigned to the device.

`MaximumAddress`

The maximum address that can be assigned to the device.

Return Value

RtlIoEncodeMemIoResource returns an NTSTATUS value. This routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The IO_RESOURCE_DESCRIPTOR structure was updated.
STATUS_UNSUCCESSFUL	The specified value for *Length* or *Alignment* could not be encoded in an **IO_RESOURCE_DESCRIPTOR** structure.
STATUS_INVALID_PARAMETER	One or more of the specified parameters were invalid.

Remarks

Addresses that are larger than 32 bits in length must satisfy certain alignment restrictions, or else the routine returns STATUS_UNSUCCESSFUL.

ADDRESS LENGTH	ALIGNMENT RESTRICTION
40 bits	Lowest 8 bits must be zero.
48 bits	Lowest 16 bits must be zero.
64 bits	Lowest 32 bits must be zero.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	Any level

See Also

IO_RESOURCE_DESCRIPTOR

RtlIoDecodeMemIoResource

RtlIsNtDdiVersionAvailable function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIsNtDdiVersionAvailable** routine determines if a specified version of the Microsoft Windows device driver interface (DDI) is available.

Syntax

```
BOOLEAN RtlIsNtDdiVersionAvailable(
  ULONG Version
);
```

Parameters

`Version`

The version of the Windows DDI that is available. The following table lists the possible values for the *Version* parameter.

CONSTANT	WINDOWS VERSION
NTDDI_WINBLUE	Windows 8.1
NTDDI_WIN8	Windows 8
NTDDI_WIN7	Windows 7
NTDDI_WS08	Windows Server 2008
NTDDI_VISTA	Windows Vista
NTDDI_WS03	Windows Server 2003
NTDDI_WINXP	Windows XP
NTDDI_WIN2K	Windows 2000

The NTDDI_*XXX* constants are defined in the Sdkddkver.h header file.

Additional NTDDI_*XXX* constants that specify service packs are available for the RtlIsServicePackVersionInstalled routine. Do not use these values for **RtlIsNtDdiVersionAvailable**.

Return Value

RtlIsNtDdiVersionAvailable returns **TRUE** if the version of the Windows operating system that is running is the same or later than the version that the *Version* parameter specifies. Otherwise, this routine returns **FALSE**.

Remarks

The **RtlIsNtDdiVersionAvailable** routine compares the version that the *Version* parameter specifies to the

version of the Windows operating system that is currently running.

Use the RtlIsServicePackVersionInstalled routine if you want to determine whether a particular service pack is installed.

The Windows kernel implements **RtlIsNtDdiVersionAvailable** only in Windows Vista and later versions of Windows. However, a compatibility library, Rtlver.lib, implements a version of **RtlIsNtDdiVersionAvailable** that runs in Windows 2000 and later versions of Windows. For kernel-mode drivers that include the Wdm.h header file, calls to **RtlIsNtDdiVersionAvailable** go to the version of this routine that is implemented in Rtlver.lib.

For more information about **RtlIsNtDdiVersionAvailable** and **RtlIsServicePackVersionInstalled**, see Writing Drivers for Different Versions of Windows.

Requirements

Minimum supported client	Available starting with Windows Vista. A compatibility library supports this routine in earlier versions of Windows (see Remarks section).
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Rtlver.lib
IRQL	PASSIVE_LEVEL

See Also

RtlIsServicePackVersionInstalled

RtlIsServicePackVersionInstalled function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIsServicePackVersionInstalled** routine determines if a specified service pack version of the Microsoft Windows device driver interface (DDI) is installed.

Syntax

```
BOOLEAN RtlIsServicePackVersionInstalled(
  ULONG Version
);
```

Parameters

`Version`

The service pack version of the Windows DDI that is available. The following table lists the possible values for *Version*.

CONSTANT	WINDOWS VERSION
NTDDI_WIN7	Windows 7 and Windows Server 2008 R2
NTDDI_WS08SP2	Windows Server 2008 with Service Pack 2 (SP2)
NTDDI_WS08	Windows Server 2008
NTDDI_VISTASP2	Windows Vista with SP2
NTDDI_VISTASP1	Windows Vista with SP1
NTDDI_VISTA	Windows Vista
NTDDI_WS03SP2	Windows Server 2003 with SP2
NTDDI_WS03SP1	Windows Server 2003 with SP1
NTDDI_WS03	Windows Server 2003
NTDDI_WINXPSP3	Windows XP with SP3
NTDDI_WINXPSP2	Windows XP with SP2
NTDDI_WINXPSP1	Windows XP with SP1
NTDDI_WINXP	Windows XP
NTDDI_WIN2KSP4	Windows 2000 with SP4

NTDDI_WIN2KSP3	Windows 2000 with SP3
NTDDI_WIN2KSP2	Windows 2000 with SP2
NTDDI_WIN2KSP1	Windows 2000 with SP1
NTDDI_WIN2K	Windows 2000

The NTDDI_*XXX* constants are defined in the Sdkddkver.h header file. The preceding table does not contain an entry for Windows Server 2008 with SP1. The first service pack to become available for Windows Server 2008 is SP2.

Return Value

RtlIsServicePackVersionInstalled returns **TRUE** if the service pack version of the Windows operating system that is running is the same or later than the version that the *Version* parameter specifies. Otherwise, this routine returns **FALSE**. **RtlIsServicePackVersionInstalled** also returns **FALSE** if the major version (for example, Windows Vista or Windows Server 2003) that *Version* specifies does not match the major version of Windows that is currently running on the computer.

Remarks

The **RtlIsServicePackVersionInstalled** routine compares the version that the *Version* parameter specifies to the version of the currently running Windows operating system.

Use the RtlIsNtDdiVersionAvailable routine to determine if a major version of Windows is running.

For more information about **RtlIsServicePackVersionInstalled** and **RtlIsNtDdiVersionAvailable**, see Header File Changes in the Windows Driver Kit.

The Windows kernel implements **RtlIsServicePackVersionInstalled** only in Windows Vista and later versions of Windows. However, a compatibility library, Rtlver.lib, implements a version of **RtlIsServicePackVersionInstalled** that runs in Windows 2000 and later versions of Windows. For kernel-mode drivers that include the Wdm.h header file, calls to **RtlIsServicePackVersionInstalled** go to the version of this routine that is implemented in Rtlver.lib.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows. A compatibility library supports this routine in earlier versions of Windows (see Remarks section).
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	Rtlver.lib
IRQL	PASSIVE_LEVEL

See Also

PsGetVersion

RtlIsNtDdiVersionAvailable

RtlVerifyVersionInfo

RtlLengthSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlLengthSecurityDescriptor** routine returns the size of a given security descriptor.

Syntax

```
NTSYSAPI ULONG RtlLengthSecurityDescriptor(
  PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Parameters

`SecurityDescriptor`

Pointer to a SECURITY_DESCRIPTOR.

Return Value

RtlLengthSecurityDescriptor returns the size in bytes of the descriptor.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCreateSecurityDescriptor

RtlSetDaclSecurityDescriptor

RtlValidSecurityDescriptor

SECURITY_DESCRIPTOR

RtlMoveMemory macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlMoveMemory** routine copies the contents of a source memory block to a destination memory block, and supports overlapping source and destination memory blocks.

Syntax

```
void RtlMoveMemory(
   Destination,
   Source,
   Length
);
```

Parameters

`Destination`

Datatype: void*. A pointer to the destination memory block to copy the bytes to.

`Source`

Datatype: const void*. A pointer to the source memory block to copy the bytes from.

`Length`

Datatype: size_t. The number of bytes to copy from the source to the destination.

Return Value

None

Remarks

The source memory block, which is defined by *Source* and *Length*, can overlap the destination memory block, which is defined by *Destination* and *Length*.

The RtlCopyMemory routine runs faster than **RtlMoveMemory**, but **RtlCopyMemory** requires that the source and destination memory blocks do not overlap.

Callers of **RtlMoveMemory** can be running at any IRQL if the source and destination memory blocks are in nonpaged system memory. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (See Remarks section)
DDI compliance rules	BufAfterReqCompletedIntIoctlA, BufAfterReqCompletedIoctlA, BufAfterReqCompletedReadA, BufAfterReqCompletedWriteA

See Also

RtlCopyMemory

RtlNumberOfClearBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlNumberOfClearBits** routine returns a count of the clear bits in a given bitmap variable.

Syntax

```
NTSYSAPI ULONG RtlNumberOfClearBits(
  PRTL_BITMAP BitMapHeader
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

Return Value

RtlNumberOfClearBits returns the number of bits that are currently clear.

Remarks

Callers of **RtlNumberOfClearBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlNumberOfClearBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlFindClearBits

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfSetBits

RtlNumberOfSetBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlNumberOfSetBits** routine returns a count of the set bits in a given bitmap variable.

Syntax

```
NTSYSAPI ULONG RtlNumberOfSetBits(
  PRTL_BITMAP BitMapHeader
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

Return Value

RtlNumberOfSetBits returns a count of the bits that are currently set.

Remarks

Callers of **RtlNumberOfSetBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlNumberOfSetBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlFindSetBits

RtlInitializeBitMap

RtlNumberOfClearBits

RtlNumberOfSetBitsUlongPtr function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlNumberOfSetBitsUlongPtr** routine returns the number of bits in the specified ULONG_PTR integer value that are set to one.

Syntax

```
NTSYSAPI ULONG RtlNumberOfSetBitsUlongPtr(
  ULONG_PTR Target
);
```

Parameters

`Target`

A ULONG_PTR integer value.

Return Value

RtlNumberOfSetBitsUlongPtr returns a count of the bits in the *Target* parameter that are set to one.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any IRQL

RtlPrefetchMemoryNonTemporal function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlPrefetchMemoryNonTemporal** routine provides a hint to the processor that a buffer should be temporarily moved into the processor cache.

Syntax

```
NTSYSAPI VOID RtlPrefetchMemoryNonTemporal(
  PVOID  Source,
  SIZE_T Length
);
```

Parameters

`Source`

A pointer to the buffer to be moved into the processor cache.

`Length`

The length of the buffer to be moved.

Return Value

None.

Remarks

You should use this routine only for a buffer that will be written to or read from exactly once. Note that **RtlPrefetchMemoryNonTemporal** is only a hint to the processor: the buffer is not guaranteed to be moved into the cache. On x86-based and x64-based systems, this routine uses the **prefetchnta** instruction.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h, Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

RtlQueryRegistryValues function

1/25/2019 • 11 minutes to read • Edit Online

The **RtlQueryRegistryValues** routine allows the caller to query several values from the registry subtree with a single call.

Syntax

```
NTSYSAPI NTSTATUS RtlQueryRegistryValues(
  ULONG                    RelativeTo,
  PCWSTR                   Path,
  PRTL_QUERY_REGISTRY_TABLE QueryTable,
  PVOID                    Context,
  PVOID                    Environment
);
```

Parameters

`RelativeTo`

Specifies whether *Path* is an absolute registry path or is relative to a predefined path as one of the following.

VALUE	MEANING
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**. (For a system process, this is **\User.Default**.)
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.

The *RelativeTo* value can be modified by bitwise-ORing it with one of the following flags.

RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use.

`Path`

Pointer to either an absolute registry path or a path relative to the known location specified by the *RelativeTo* parameter. Note that the names of keys in such a path must be known to the caller, including the last key in the path. If the RTL_REGISTRY_HANDLE flag is specified, this parameter is a registry handle for an already opened key to be queried directly.

`QueryTable`

Pointer to a table of one or more value names and subkey names in which the caller is interested. Each table entry contains the address of a caller-supplied QueryRoutine function that will be called for each value name that exists in the registry. The table must be terminated with a **NULL** table entry, which is a table entry with a **NULL** **QueryRoutine** member and a **NULL** **Name** member. The structure for query table entries is defined as follows:

```
typedef struct _RTL_QUERY_REGISTRY_TABLE {
    PRTL_QUERY_REGISTRY_ROUTINE QueryRoutine;
    ULONG Flags;
    PWSTR Name;
    PVOID EntryContext;
    ULONG DefaultType;
    PVOID DefaultData;
    ULONG DefaultLength;
} RTL_QUERY_REGISTRY_TABLE, *PRTL_QUERY_REGISTRY_TABLE;
```

If the caller allocates storage for the query table pointed to by the *QueryTable* parameter, the caller is responsible for releasing this storage after the **RtlQueryRegistryValues** call returns.

QueryRoutine

The address of a *QueryRoutine* function that is called with the name, type, data, and data length of a registry value. If this member and the **Name** member are both **NULL**, it marks the end of the table.

A *QueryRoutine* function is declared as follows:

```
NTSTATUS
QueryRoutine (
    IN PWSTR ValueName,
    IN ULONG ValueType,
    IN PVOID ValueData,
    IN ULONG ValueLength,
    IN PVOID Context,
    IN PVOID EntryContext
    );
```

For more information, see QueryRoutine.

Flags

Flags to control how the remaining members of the **RTL_QUERY_REGISTRY_TABLE** structure are to be interpreted. The following flag bits are defined for this member.

VALUE	MEANING

RTL_QUERY_REGISTRY_SUBKEY	The **Name** of this table entry is another path to a registry key, and all following table entries are for that key rather than the key specified by the *Path* parameter. This change in focus lasts until the end of the table or until another RTL_REGISTRY_SUBKEY or RTL_QUERY_REGISTRY_TOPKEY entry is seen. Each such entry must specify a path that is relative to the *Path* specified in the call to **RtlQueryRegistryValues**.
RTL_QUERY_REGISTRY_TOPKEY	Resets the current registry key handle to the original one specified by the *RelativeTo* and *Path* parameters. This is useful for getting back to the original node after descending into subkeys with the RTL_QUERY_REGISTRY_SUBKEY flag.
RTL_QUERY_REGISTRY_REQUIRED	Specifies that this registry value must exist if **DefaultType** = REG_NONE; otherwise, if it is not found, **RtlQueryRegistryValues** immediately exits with a status code of STATUS_OBJECT_NAME_NOT_FOUND. This exit occurs if the **Name** member is **NULL** and the current key has no subkeys, or if **Name** specifies a nonexistent subkey. (If this flag is not specified, then when no match is found for a non-**NULL** **Name**, the routine uses the **DefaultValue** member as the value. When **Name** is **NULL** and the current key has no subkeys, the routine simply skips that table entry.)
RTL_QUERY_REGISTRY_NOVALUE	Specifies that even though there is no **Name** for this table entry, all the caller wants is a callback: that is, the caller does not want to enumerate all the values under the current key. *QueryRoutine* is called with **NULL** for *ValueData*, REG_NONE for *ValueType*, and zero for *ValueLength*.
RTL_QUERY_REGISTRY_NOEXPAND	For a registry value of type REG_EXPAND_SZ or REG_MULTI_SZ, this flag overrides the default behavior, which is to preprocess the registry value before calling the QueryRoutine routine. By default, **RtlQueryRegistryValues** expands environment variable references in REG_EXPAND_SZ values, and enumerates each null-terminated string in a REG_MULTI_SZ value in a separate *QueryRoutine* call, so that the strings are presented as REG_SZ values that have the same *ValueName*. If this flag is set, *QueryRoutine* receives the raw REG_EXPAND_SZ or REG_MULTI_SZ value from the registry. For more information about the data formats for these values, see KEY_VALUE_BASIC_INFORMATION.
RTL_QUERY_REGISTRY_DIRECT	The **QueryRoutine** member is not used (and must be **NULL**), and the *EntryContext* points to the buffer to store the value. If the caller sets this flag, the caller should additionally set the RTL_QUERY_REGISTRY_TYPECHECK flag to guard against buffer overflow. For more information, see the Remarks section.
RTL_QUERY_REGISTRY_TYPECHECK	Use this flag with the RTL_QUERY_REGISTRY_DIRECT flag to verify that the REG_*XXX* type of the stored registry value matches the type expected by the caller. If the types do not match, the call fails. For more information, see the Remarks section.
RTL_QUERY_REGISTRY_DELETE	This flag is used to delete value keys after they have been queried.

Starting with Windows 2000, inbox support is provided for all flag bits in the preceding table, with the exception of

RTL_QUERY_REGISTRY_TYPECHECK. Inbox support for RTL_QUERY_REGISTRY_TYPECHECK is available starting with Windows 8. For earlier versions of Windows, support for RTL_QUERY_REGISTRY_TYPECHECK is provided through Windows Update. For more information, see Remarks.

Name

This is the name of a Value that the caller queried. If **Name** is **NULL**, the *QueryRoutine* function specified for this table entry is called for all values associated with the current registry key. If the RTL_QUERY_REGISTRY_DIRECT flag is set, a non-**NULL** value for **Name** must be provided.

EntryContext

If the RTL_QUERY_REGISTRY_DIRECT flag is set, this is a pointer to the buffer to store the result of the query operation for this key. Otherwise, this value is passed as the *EntryContext* parameter of *QueryRoutine*.

DefaultType

The least significant byte of this member specifies the REG_*XXX* type of the data to be returned, if no matching key is found and the RTL_QUERY_REGISTRY_REQUIRED flag is not specified. Specify REG_NONE for no default type. If the RTL_QUERY_REGISTRY_TYPECHECK flag is set, the most significant byte of this member specifies the REG_*XXX* type of the stored registry value that the caller expects. Bits 8 to 23 of this member are reserved and should be zero.

DefaultData

A pointer to the default value to be returned if no matching key is found and the RTL_QUERY_REGISTRY_REQUIRED flag is not specified. This member is ignored if **DefaultType** = REG_NONE. Otherwise, the type of data pointed to by **DefaultData** should conform to the registry value type specified by the **DefaultType** member. For more information registry value types, see the definition of the *Type* parameter in KEY_VALUE_BASIC_INFORMATION.

DefaultLength

Specifies the length, in bytes, of the **DefaultData** member. If **DefaultType** is REG_SZ, REG_EXPAND_SZ, or REG_MULTI_SZ, callers can optionally specify zero to indicate **RtlQueryRegistryValues** should compute the length based on the default data value. If **DefaultType** = REG_NONE, this member is ignored.

`Context`

Specifies the value passed as the *Context* parameter of a *QueryRoutine* function each time it is called.

`Environment`

Pointer to the environment used when expanding variable values in REG_EXPAND_SZ registry values, or a **NULL** pointer (optional).

Return Value

RtlQueryRegistryValues returns an NTSTATUS code. The possible return values include:

RETURN CODE	DESCRIPTION
STA **TUS** **_SU** **CCE** **SS**	The entire query table was processed successfully.

STATUS_INVALID_PARAMETER	Processing of the query table terminated with an invalid table entry. A table entry can be invalid if the specified flags require the **QueryRoutine** or **Name** member to be non-**NULL**, but a **NULL** value was provided.
STATUS_OBJECT_NAME_NOT_FOUND	The *Path* parameter does not match a valid key, or processing of the query table terminated with an entry with the RTL_QUERY_REGISTRY_REQUIRED flag set and no matching key is found. This occurs if the **Name** member is **NULL** and the current key has no subkeys, or if **Name** specifies a nonexistent subkey.
STATUS_BUFFER_TOO_SMALL	The RTL_QUERY_REGISTRY_DIRECT flag is set, and the buffer specified by **EntryContext** is too small to hold the key value data.
STATUS_OBJECT_TYPE_MISMATCH	The RTL_QUERY_REGISTRY_TYPECHECK flag is set and the type of the stored registry value does not match the type expected by the caller.

RtlQueryRegistryValues also terminates processing of the table if the *QueryRoutine* function for a table entry returns an NTSTATUS error code, and returns that error code as its result. (With one exception: If *QueryRoutine* returns STATUS_BUFFER_TOO_SMALL, the error code is ignored.)

Remarks

The caller specifies an initial key path and a table. The table contains one or more entries that describe the key values and subkey names in which the caller is interested. The table is terminated by an entry with a **NULL QueryRoutine** member and a **NULL Name** member. The table must be allocated from nonpaged pool.

> **Note** Kernel-mode drivers must specify the RTL_QUERY_REGISTRY_NOEXPAND flag to prevent calling environment variable routines. These routines are unsafe, so kernel-mode drivers should not use them.

Caution If you use the RTL_QUERY_REGISTRY_DIRECT flag, an untrusted user-mode application may be able to cause a buffer overflow. A buffer overflow can occur if a driver uses this flag to read a registry value to which the wrong type is assigned. In all cases, a driver that uses the RTL_QUERY_REGISTRY_DIRECT flag should additionally use the RTL_QUERY_REGISTRY_TYPECHECK flag to prevent such overflows.

If the RTL_QUERY_REGISTRY_TYPECHECK flag is set in a table entry, the caller must specify the expected REG_*XXX* type in the 8 most significant bits (MSBs) of the 32-bit **DefaultType** member of the table entry. As shown in the following code example, the RTL_QUERY_REGISTRY_TYPECHECK_SHIFT constant, which is defined to be 24, can be used as the shift count required to place the expected REG_*XXX* type in the 8 MSBs of the **DefaultType** member.

C++

```
RTL_QUERY_REGISTRY_TABLE QueryRegTable[2];
...
QueryRegTable[0].DefaultType = (REG_SZ << RTL_QUERY_REGISTRY_TYPECHECK_SHIFT) | REG_NONE;
...
QueryRegTable[1].DefaultType = (REG_DWORD << RTL_QUERY_REGISTRY_TYPECHECK_SHIFT) | REG_NONE;
...
```

Starting with Windows 8, if an **RtlQueryRegistryValues** call accesses an untrusted hive, and the caller sets the RTL_QUERY_REGISTRY_DIRECT flag for this call, the caller must additionally set the RTL_QUERY_REGISTRY_TYPECHECK flag. A violation of this rule by a call from user mode causes an exception. A violation of this rule by a call from kernel mode causes a 0x139 bug check (KERNEL_SECURITY_CHECK_FAILURE).

Only system hives are trusted. An **RtlQueryRegistryValues** call that accesses a system hive does not cause an exception or a bug check if the RTL_QUERY_REGISTRY_DIRECT flag is set and the RTL_QUERY_REGISTRY_TYPECHECK flag is not set. However, as a best practice, the RTL_QUERY_REGISTRY_TYPECHECK flag should always be set if the RTL_QUERY_REGISTRY_DIRECT flag is set.

Similarly, in versions of Windows before Windows 8, as a best practice, an **RtlQueryRegistryValues** call that sets the RTL_QUERY_REGISTRY_DIRECT flag should additionally set the RTL_QUERY_REGISTRY_TYPECHECK flag. However, failure to follow this recommendation does not cause an exception or a bug check.

The following is a list of system hives:

- \REGISTRY\MACHINE\HARDWARE
- \REGISTRY\MACHINE\SOFTWARE
- \REGISTRY\MACHINE\SYSTEM
- \REGISTRY\MACHINE\SECURITY
- \REGISTRY\MACHINE\SAM

Support for the RTL_QUERY_REGISTRY_TYPECHECK flag is available through Windows Update for Windows 7, Windows Vista, Windows Server 2003, and Windows XP. For more information about this update, see Vulnerabilities in Windows Kernel Could Allow Elevation of Privilege (2393802). In versions of these operating systems that do not have this update, the caller can use the RTL_QUERY_REGISTRY_TYPECHECK flag. However, this flag is ignored by the **RtlQueryRegistryValues** routine.

Starting with Windows Driver Kit (WDK) 8, the RTL_QUERY_REGISTRY_TYPECHECK flag is defined in the Wdm.h header file as follows:

```
#define RTL_QUERY_REGISTRY_TYPECHECK 0x00000100
```

If an entry does not specify the RTL_QUERY_REGISTRY_DIRECT flag, **RtlQueryRegistryValues** uses the specified *QueryRoutine* function to report the value name, type, data, and data length, in bytes, to the caller. If the **Name** member of the entry is **NULL**, **RtlQueryRegistryValues** reports every direct subkey of the key. If the key type is REG_MULTI_SZ and the RTL_QUERY_REGISTRY_NOEXPAND flag not is specified, the routine calls *QueryRoutine* separately for each individual string; otherwise the routine reports it as a single value. If an entry specifies the RTL_QUERY_REGISTRY_DIRECT flag, **RtlQueryRegistryValues** stores the value of the key in the buffer pointed to by the entry's **EntryContext** member. The format of the returned data is as follows.

KEY DATA TYPE	HOW DATA IS RETURNED
A null-terminated Unicode string (such as REG_SZ, REG_EXPAND_SZ).	**EntryContext** must point to an initialized UNICODE_STRING structure. If the **Buffer** member of **UNICODE_STRING** is **NULL**, the routine allocates storage for the string data. Otherwise, it stores the string data in the buffer that **Buffer** points to.
REG_MULTI_SZ	You must specify the RTL_QUERY_REGISTRY_NOEXPAND flag for this key data type. **EntryContext** points to an initialized **UNICODE_STRING** structure. The routine stores the key value as a single string value. Each individual component within the string is terminated by a zero. If the **Buffer** member of **UNICODE_STRING** is **NULL**, the routine allocates storage for the string data. Otherwise, it stores the string data in the buffer that **Buffer** points to.
Nonstring data with size, in bytes, <= **sizeof**(ULONG)	The value is stored in the memory location specified by **EntryContext**.
Nonstring data with size, in bytes, > **sizeof**(ULONG)	The buffer pointed to by **EntryContext** must begin with a signed LONG value. The magnitude of the value must specify the size, in bytes, of the buffer. If the sign of the value is negative, **RtlQueryRegistryValues** will only store the data of the key value. Otherwise, it will use the first ULONG in the buffer to record the value length, in bytes, the second ULONG to record the value type, and the rest of the buffer to store the value data.

If an error occurs at any stage of processing of the query table, **RtlQueryRegistryValues** stops processing the table and returns the error status.

See ZwSetValueKey for a description of the possible REG_*XXX* values.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	Ntoskrnl.lib
DLL	Ntoskrnl.exe
IRQL	PASSIVE_LEVEL

See Also

QueryRoutine

RtlZeroMemory

UNICODE_STRING

ZwEnumerateKey

ZwEnumerateValueKey

ZwSetValueKey

RtlSecureZeroMemory function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlSecureZeroMemory** routine fills a block of memory with zeros in a way that is guaranteed to be secure.

Syntax

```
PVOID RtlSecureZeroMemory(
  PVOID  ptr,
  SIZE_T cnt
);
```

Parameters

`ptr`

Pointer to the memory buffer to be filled with zeros.

`cnt`

Specifies the number of bytes to be filled with zeros.

Return Value

None

Remarks

The effect of **RtlSecureZeroMemory** is identical to that of RtlZeroMemory, except that it is guaranteed to zero the memory location, even if it is not subsequently written to. (The compiler can optimize away a call to **RtlZeroMemory**, if it determines that the caller does not access that memory range again.)

Use **RtlSecureZeroMemory** to guarantee that sensitive information has been zeroed out. For example, suppose that a function uses a local array variable to store password information. Once the function exits, the password information can remain in the same memory location unless zeroed out by **RtlSecureZeroMemory**.

RtlSecureZeroMemory is slower than **RtlZeroMemory**; therefore, if security is not an issue, use **RtlZeroMemory** instead.

Callers of **RtlSecureZeroMemory** can be running at any IRQL if the *ptr* block is in nonpaged pool. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows. (Because the routine is declared inline, the body of the routine can be included in earlier versions of the operating system.)

Target Platform	Desktop
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
IRQL	Any level (See Remarks section)

See Also

RtlFillMemory

RtlZeroMemory

RtlSetAllBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlSetAllBits** routine sets all bits in a given bitmap variable.

Syntax

```
NTSYSAPI VOID RtlSetAllBits(
  PRTL_BITMAP BitMapHeader
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

Return Value

None

Remarks

Callers of **RtlSetAllBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlSetAllBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlAreBitsSet

RtlFindClearBits

RtlFindClearBitsAndSet

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfSetBits

RtlSetBits

RtlSetBit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlSetBit** routine sets the specified bit in a bitmap to one.

Syntax

```
NTSYSAPI VOID RtlSetBit(
  PRTL_BITMAP BitMapHeader,
  ULONG       BitNumber
);
```

Parameters

`BitMapHeader`

Pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`BitNumber`

Specifies the zero-based index of the bit within the bitmap. The routine sets this bit to one.

Return Value

None

Remarks

Callers of **RtlSetBit** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlSetBit** can be called at any IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlInitializeBitMap

RtlSetBits function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlSetBits** routine sets all bits in a given range of a given bitmap variable.

Syntax

```
NTSYSAPI VOID RtlSetBits(
  PRTL_BITMAP BitMapHeader,
  ULONG       StartingIndex,
  ULONG       NumberToSet
);
```

Parameters

`BitMapHeader`

A pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`StartingIndex`

Specifies the start of the bit range to be set. This is a zero-based value indicating the position of the first bit in the range.

`NumberToSet`

Specifies how many bits to set.

Return Value

None

Remarks

RtlSetBits simply returns control if the input *NumberToSet* is zero. *StartingIndex* plus *NumberToSet*must be less than or equal to *BitMapHeader*->**SizeOfBitMap**.

Callers of **RtlSetBits** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlSetBits** can be called at any IRQL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL (See Remarks section)"

See Also

RTL_BITMAP

RtlAreBitsClear

RtlFindClearBitsAndSet

RtlFindClearRuns

RtlFindFirstRunClear

RtlFindLastBackwardRunClear

RtlFindLongestRunClear

RtlFindNextForwardRunClear

RtlInitializeBitMap

RtlNumberOfClearBits

RtlSetAllBits

RtlSetDaclSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlSetDaclSecurityDescriptor** routine sets the DACL information of an absolute-format security descriptor, or if there is already a DACL present in the security descriptor, it is superseded.

Syntax

```
NTSYSAPI NTSTATUS RtlSetDaclSecurityDescriptor(
  PSECURITY_DESCRIPTOR SecurityDescriptor,
  BOOLEAN              DaclPresent,
  PACL                 Dacl,
  BOOLEAN              DaclDefaulted
);
```

Parameters

`SecurityDescriptor`

A pointer to a SECURITY_DESCRIPTOR structure. This structure is the security descriptor to which the DACL is to be applied.

`DaclPresent`

Whether to indicate that a DACL is present in the security descriptor. If this parameter is **FALSE**, **RtlSetDaclSecurityDescriptor** sets the SE_DACL_PRESENT control flag in the security descriptor to **FALSE**. In this case, the remaining optional parameters (*Dacl* and *DaclDefaulted*) are ignored. If *DaclPresent* is **TRUE**, the SE_DACL_PRESENT flag in the security descriptor is set to **TRUE** and the remaining optional parameters are not ignored. For more information about the SE_DACL_PRESENT flag, see SECURITY_DESCRIPTOR_CONTROL.

`Dacl`

A pointer to the DACL for the security descriptor. If this parameter is **NULL**, the DACL pointer in the security descriptor is set to **NULL**. A **NULL** DACL pointer unconditionally grants all access to an object and is not the same as an empty DACL. An empty DACL denies all access to an object. If *Dacl* is non-**NULL**, the ACL structure supplied by the caller is referenced by, but not copied into, the security descriptor. The caller can allocate the **ACL** structure from paged system memory, and can call the RtlCreateAcl routine to initialize the structure.

`DaclDefaulted`

Whether the DACL was supplied by some default mechanism, or was explicitly specified by the caller. If this parameter is **TRUE**, the DACL was supplied by a default mechanism. If *DaclDefaulted* is **FALSE**, the caller explicitly specified the DACL. **RtlSetDaclSecurityDescriptor** copies the value specified for this parameter to the SE_DACL_DEFAULTED control flag in the security descriptor. For more information about the SE_DACL_DEFAULTED flag, see SECURITY_DESCRIPTOR_CONTROL.

Return Value

RtlSetDaclSecurityDescriptor can return one of the following status values.

RETURN CODE	DESCRIPTION

STA TUS _SU CCE SS	The call completed successfully.
STA TUS _UN KN OW N_R EVIS ION	The revision of the security descriptor is unknown.
STA TUS _IN VAL ID_S ECU RIT Y_D ESC R	The security descriptor is not an absolute format security descriptor.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

ACL

RtlCreateAcl

RtlCreateSecurityDescriptor

RtlLengthSecurityDescriptor

RtlValidSecurityDescriptor

SECURITY_DESCRIPTOR

SECURITY_DESCRIPTOR_CONTROL

RtlStringFromGUID function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlStringFromGUID** routine converts a given GUID from binary format into a Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlStringFromGUID(
  REFGUID         Guid,
  PUNICODE_STRING GuidString
);
```

Parameters

`Guid`

Specifies the binary-format GUID to convert.

`GuidString`

Pointer to a caller-supplied variable in which a pointer to the converted GUID string is returned. **RtlStringFromGUID** allocates the buffer space for the string, which the caller must free by calling **RtlFreeUnicodeString**.

Return Value

If the conversion succeeds, **RtlStringFromGUID** returns STATUS_SUCCESS. Otherwise, no storage was allocated, nor was the conversion performed.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlGUIDFromString

RtlTestBit function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlTestBit** routine returns the value of a bit in a bitmap.

Syntax

```
NTSYSAPI BOOLEAN RtlTestBit(
  PRTL_BITMAP BitMapHeader,
  ULONG       BitNumber
);
```

Parameters

`BitMapHeader`

Pointer to the RTL_BITMAP structure that describes the bitmap. This structure must have been initialized by the RtlInitializeBitMap routine.

`BitNumber`

Specifies the zero-based index of the bit within the bitmap. The routine returns the value of this bit.

Return Value

RtlTestBit returns the value of the bit that the *BitNumber* parameter points to.

Remarks

Callers of **RtlTestBit** must be running at IRQL <= APC_LEVEL if the memory that contains the bitmap variable is pageable or the memory at *BitMapHeader* is pageable. Otherwise, **RtlTestBit** can be called at any IRQL.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL (see Remarks section)"

See Also

RTL_BITMAP

RtlInitializeBitMap

RtlTimeFieldsToTime function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlTimeFieldsToTime** routine converts **TIME_FIELDS** information to a system time value.

Syntax

```
NTSYSAPI BOOLEAN RtlTimeFieldsToTime(
  PTIME_FIELDS   TimeFields,
  PLARGE_INTEGER Time
);
```

Parameters

`TimeFields`

Pointer to the following structure, containing the time information to be converted:

```
typedef struct TIME_FIELDS {
    CSHORT Year;
    CSHORT Month;
    CSHORT Day;
    CSHORT Hour;
    CSHORT Minute;
    CSHORT Second;
    CSHORT Milliseconds;
    CSHORT Weekday;
} TIME_FIELDS;
```

Year

Specifies a value from 1601 on.

Month

Specifies a value from 1 to 12.

Day

Specifies a value from 1 to 31.

Hour

Specifies a value from 0 to 23.

Minute

Specifies a value from 0 to 59.

Second

Specifies a value from 0 to 59.

Milliseconds

Specifies a value from 0 to 999.

Weekday

Specifies a value from 0 to 6 (Sunday to Saturday).

`Time`

Pointer to a buffer, which is to contain the converted system time value as a large integer.

Return Value

RtlTimeFieldsToTime returns **TRUE** if the input *TimeFields* data was successfully converted.

Remarks

RtlTimeFieldsToTime ignores the **Weekday** value in *TimeFields*.

Callers of **RtlTimeFieldsToTime** can be running at any IRQL if both input buffers are resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (See Remarks section)

See Also

ExLocalTimeToSystemTime

ExSystemTimeToLocalTime

KeQuerySystemTime

RtlTimeToTimeFields

RtlTimeToTimeFields function

1/25/2019 • 2 minutes to read • Edit Online

The RtlTimeToTimeFields routine converts system time into a **TIME_FIELDS** structure.

Syntax

```
NTSYSAPI VOID RtlTimeToTimeFields(
  PLARGE_INTEGER Time,
  PTIME_FIELDS   TimeFields
);
```

Parameters

`Time`

Pointer to a buffer containing the absolute system time as a large integer, accurate to 100-nanosecond resolution.

`TimeFields`

Pointer to a caller-allocated buffer, which must be at least **sizeof**(**TIME_FIELDS**), to contain the returned information.

Return Value

None

Remarks

Callers of **RtlTimeToTimeFields** can be running at any IRQL if both input buffers are resident.

For more info about the **TIME_FIELDS** structure, see RtlTimeFieldsToTime function.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	Any level (See Remarks section)

See Also

ExLocalTimeToSystemTime

ExSystemTimeToLocalTime

KeQuerySystemTime

RtlTimeFieldsToTime

RtlUlongByteSwap macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUlongByteSwap** routine reverses the ordering of the four bytes in a 32-bit unsigned integer value.

Syntax

```
void RtlUlongByteSwap(
   _x
);
```

Parameters

`_x`

A ULONG value to convert to a byte-swapped version.

Return Value

The byte-swapped version of the Source input parameter value.

Remarks

For example, if the *Source* parameter value is 0x12345678, the routine returns 0x78563412.

A typical use of this routine is to convert a ULONG value from little-endian byte format to big-endian byte format, and vice versa.

Use this routine instead of **ntohl** or **htonl**.

To reverse the ordering of bytes in a USHORT value, use the RtlUshortByteSwap routine. To reverse ordering of bytes in a ULONGLONG value, use the RtlUlonglongByteSwap routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

RtlUlonglongByteSwap

RtlUshortByteSwap

RtlUlonglongByteSwap macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUlonglongByteSwap** routine reverses the ordering of the eight bytes in a 64-bit unsigned integer value.

Syntax

```
void RtlUlonglongByteSwap(
   _x
);
```

Parameters

`_x`

A ULONGLONG value to convert to a byte-swapped version.

Return Value

The byte-swapped version of the input parameter value.

Remarks

For example, if the *Source* parameter value is 0x0123456789abcdef, the routine returns 0xefcdab8967452301.

A typical use of this routine is to convert a ULONGLONG value from little-endian byte format to big-endian byte format, and vice versa.

To reverse the ordering of bytes in a USHORT value, use the RtlUshortByteSwap routine. To reverse ordering of bytes in a ULONG value, use the RtlUlongByteSwap routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

RtlUlongByteSwap

RtlUshortByteSwap

RtlUnicodeStringToAnsiSize macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringToAnsiSize** routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.

Syntax

```
void RtlUnicodeStringToAnsiSize(
   STRING
);
```

Parameters

`STRING`

Pointer to the Unicode string for which to compute the number of bytes required for an equivalent null-terminated ANSI string.

Return Value

If the Unicode string can be translated into an ANSI string using the current system locale information, **RtlUnicodeStringToAnsiSize** returns the number of bytes required for an equivalent null-terminated ANSI string. Otherwise, **RtlUnicodeStringToAnsiSize** returns zero.

Remarks

The Unicode string is interpreted for the current system locale.

RtlUnicodeStringToAnsiSize performs the same operation as RtlxUnicodeStringToAnsiSize, but executes faster if the system does not use multibyte code pages.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlxUnicodeStringToAnsiSize

RtlUnicodeStringToAnsiString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringToAnsiString** routine converts a given Unicode string into an ANSI string.

Syntax

```
NTSYSAPI NTSTATUS RtlUnicodeStringToAnsiString(
  PANSI_STRING     DestinationString,
  PCUNICODE_STRING SourceString,
  BOOLEAN          AllocateDestinationString
);
```

Parameters

`DestinationString`

Pointer to an ANSI_STRING structure to hold the converted ANSI string. If *AllocateDestinationString* is **TRUE**, the routine allocates a new buffer to hold the string data, and updates the **Buffer** member of *DestinationString* to point to the new buffer. Otherwise, the routine uses the currently specified buffer to hold the string.

`SourceString`

Pointer to the Unicode source string to be converted to ANSI.

`AllocateDestinationString`

TRUE if this routine is to allocate the buffer space for the *DestinationString*. If it does, the buffer must be deallocated by calling RtlFreeAnsiString.

Return Value

If the conversion succeeds, **RtlUnicodeStringToAnsiString** returns STATUS_SUCCESS. Otherwise, no storage was allocated, and no conversion was done.

Remarks

The translation is done in accord with the current system-locale information.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ANSI_STRING

RtlAnsiStringToUnicodeString

RtlFreeAnsiString

UNICODE_STRING

RtlUnicodeStringToInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringToInteger** routine converts a Unicode string representation of a number to the equivalent integer value.

Syntax

```
NTSYSAPI NTSTATUS RtlUnicodeStringToInteger(
  PCUNICODE_STRING String,
  ULONG            Base,
  PULONG           Value
);
```

Parameters

`String`

A pointer to a UNICODE_STRING structure that contains the number representation to convert to the equivalent integer value.

`Base`

A numeric value that indicates the base (or radix) of the number that the Unicode string represents. This parameter value is optional and can be set to zero.

If *Base* is zero, **RtlUnicodeStringToInteger** checks the prefix of the Unicode string to determine the base of the number:

- If the prefix is "0x", **RtlUnicodeStringToInteger** interprets the number in the string as a hexadecimal integer.
- If the prefix is "0o", **RtlUnicodeStringToInteger** interprets the number in the string as an octal integer.
- If the prefix is "0b", **RtlUnicodeStringToInteger** interprets the number in the string as a binary integer.

If the Unicode string does not contain any of these prefixes, **RtlUnicodeStringToInteger** treats the string as a base-10 integer.

`Value`

A pointer to a ULONG variable to which **RtlUnicodeStringToInteger** writes the integer value that results from conversion of the Unicode string.

Return Value

If the conversion is successful, the **RtlUnicodeStringToInteger** routine returns STATUS_SUCCESS and sets **Value* to the integer value represented by the number in the Unicode string. If the string is not empty, but does not start with a valid number representation, the routine returns STATUS_SUCCESS and sets **Value* to zero. If the string is empty, the routine fails and returns STATUS_INVALID_PARAMETER.

Remarks

This routine skips any white space at the start of the input string to find the start of the number.

If the first non-white space character in the string is a hyphen (-), the integer value written to **Value* is negative; otherwise, if the first character is a "+" or there is no sign character, the integer value written to **Value* is positive.

If the string does not contain a valid number, or if the first digit in the string is preceded by a non-white space character other than '+' or '-', the routine sets the output value to zero and returns STATUS_SUCCESS.

A substring that contains one or more valid digits is terminated by any character that is not a valid digit. For example, if *Base* = 2, valid digits are '0' and '1'. If *Base* = 8, valid digits are '0' to '7'. If *Base* = 10, valid digits are '0' to '9'. If *Base* = 16, valid digits are '0' to '9', 'a' to 'f', and 'A' to 'F'.

The following table contains examples of output values that result from various combinations of input strings and *Base* parameter values.

INPUT STRING	BASE	OUTPUT VALUE
"123"	10	123
" -345"	10	-345
"xyz"	10	0
" +678abc"	10	678
" +678abc"	16	6785724
"007"	10	7
"789"	8	7
"FGH"	16	15
" "	10	0

A related routine, RtlIntegerToUnicodeString, converts an integer value to the equivalent Unicode string representation.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlIntegerToUnicodeString

UNICODE_STRING

RtlUnicodeToUTF8N function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUnicodeToUTF8N** routine converts a Unicode string to a UTF-8 string.

Syntax

```
NTSYSAPI NTSTATUS RtlUnicodeToUTF8N(
  PCHAR  UTF8StringDestination,
  ULONG  UTF8StringMaxByteCount,
  PULONG UTF8StringActualByteCount,
  PCWCH  UnicodeStringSource,
  ULONG  UnicodeStringByteCount
);
```

Parameters

`UTF8StringDestination`

A pointer to a caller-allocated destination buffer into which the routine writes the UTF-8 output string. If this parameter is **NULL**, the routine writes the required size of the output buffer to **UTF8StringActualByteCount*.

`UTF8StringMaxByteCount`

Specifies the maximum number of bytes that the routine can write to the buffer that *UTF8StringDestination* points to. If *UTF8StringDestination* = **NULL**, set *UTF8StringMaxByteCount* = 0.

`UTF8StringActualByteCount`

A pointer to a location into which the routine writes the actual number of bytes that it has written to the buffer that *UTF8StringDestination* points to. If *UTF8StringDestination* is non-**NULL**, this count never exceeds the value of *UTF8StringMaxByteCount*. If *UTF8StringDestination* is **NULL**, this count is the number of bytes that are required to contain the entire output string.

`UnicodeStringSource`

A pointer to the Unicode source string.

`UnicodeStringByteCount`

Specifies the number of bytes in the Unicode source string that the *UnicodeStringSource* parameter points to.

Return Value

RtlUnicodeToUTF8N returns STATUS_SUCCESS if the call is successful and all Unicode character codes in the input string were converted to the corresponding UTF-8 character codes in the output string. It returns STATUS_SOME_NOT_MAPPED if the call is successful but one or more input characters were invalid and were replaced by the Unicode replacement character, U+FFFD, before they were converted to UTF-8. Possible error return values include the following error codes:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_TOO_SMALL	The *UTF8StringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string.
STATUS_INVALID_PARAMETER	The *UTF8StringDestination* and *UTF8StringActualByteCount* parameters are both **NULL**.
STATUS_INVALID_PARAMETER_4	The *UnicodeStringSource* parameter is **NULL**.
STATUS_INVALID_PARAMETER_5	*UnicodeStringByteCount* is not an integer multiple of **sizeof**(WCHAR).

Remarks

The UTF-8 output string is null-terminated only if the Unicode input string is null-terminated.

The routine returns STATUS_BUFFER_TOO_SMALL if the *UTF8StringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string. In this case, the routine writes as many UTF-8 characters as will fit in the buffer, and the **UTF8StringActualByteCount* value specifies the number of valid bytes that the routine has written to the buffer. The partial string that is contained in the output buffer might not include a terminating null character.

You can make an initial call to **RtlUnicodeToUTF8N** to obtain the required output buffer size, and then call **RtlUnicodeToUTF8N** again to obtain the Unicode output string. In the initial call, set *UTF8StringDestination* = **NULL** and *UTF8StringMaxByteCount* = 0, and the routine will write the required buffer size to **UTF8StringActualByteCount*. Next, allocate a buffer of the required size and call **RtlUnicodeToUTF8N** a second time to obtain the UTF-8 output string.

RtlUnicodeToUTF8N supports Unicode surrogate pairs. However, a surrogate leading word value that is not followed by a trailing word value, or a trailing word value that is not preceded by a leading word value, is not recognized as a valid character code and is replaced by the Unicode replacement character, U+FFFD.

RtlUnicodeToUTF8N continues to convert the input string to an output string until it reaches the end of the source buffer or the end of the destination buffer, whichever occurs first. The routine converts any null characters in the input string to null characters in the output string. If the input string contains a terminating null character, but the null character is not located at the end of the source buffer, the routine continues past the terminating null character until it reaches the end of the available buffer space.

The RtlUTF8ToUnicodeN routine converts a UTF-8 string to a Unicode string.

You can use **RtlUnicodeToUTF8N** and **RtlUTF8ToUnicode** routines to perform a lossless conversion of valid text strings between the Unicode and UTF-8 formats. However, strings that have arbitrary data values are likely to violate the Unicode rules for encoding surrogate pairs, and any information that is contained in the invalid values in an input string is lost and cannot be recovered from the resulting output string.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntifs.h, Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlUTF8ToUnicodeN

RtlUpcaseUnicodeChar function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUpcaseUnicodeChar** routine converts the specified Unicode character to uppercase.

Syntax

```
NTSYSAPI WCHAR RtlUpcaseUnicodeChar(
  WCHAR SourceCharacter
);
```

Parameters

`SourceCharacter`

Specifies the character to convert.

Return Value

RtlUpcaseUnicodeChar returns the uppercase version of the specified Unicode character.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	"<= APC_LEVEL"

See Also

RtlUpcaseUnicodeString

RtlUpperChar

RtlUshortByteSwap macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUshortByteSwap** routine reverses the ordering of the two bytes in a 16-bit unsigned integer value.

Syntax

```
void RtlUshortByteSwap(
   _x
);
```

Parameters

`_x`

A USHORT value to convert to a byte-swapped version.

Return Value

The byte-swapped version of the input parameter value.

Remarks

For example, if the *Source* parameter value is 0x1234, the routine returns 0x3412.

A typical use of this routine is to convert a USHORT value from little-endian byte format to big-endian byte format, and vice versa.

Use this routine instead of **ntohs** or **htons**.

To reverse the ordering of bytes in a ULONG value, use the RtlUlongByteSwap routine. To reverse ordering of bytes in a ULONGLONG value, use the RtlUlonglongByteSwap routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

RtlUlongByteSwap

RtlUlonglongByteSwap

RtlUTF8ToUnicodeN function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlUTF8ToUnicodeN** routine converts a UTF-8 string to a Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlUTF8ToUnicodeN(
  PWSTR  UnicodeStringDestination,
  ULONG  UnicodeStringMaxByteCount,
  PULONG UnicodeStringActualByteCount,
  PCCH   UTF8StringSource,
  ULONG  UTF8StringByteCount
);
```

Parameters

`UnicodeStringDestination`

A pointer to a caller-allocated destination buffer into which the routine writes the Unicode output string. If this parameter is **NULL**, the routine writes the required size of the output buffer to **UnicodeStringActualByteCount*.

`UnicodeStringMaxByteCount`

Specifies the maximum number of bytes that the routine can write to the buffer that *UnicodeStringDestination* points to. If *UnicodeStringDestination* = **NULL**, set *UnicodeStringMaxByteCount* = 0.

`UnicodeStringActualByteCount`

A pointer to a location into which the routine writes the actual number of bytes that the routine has written to the buffer that *UnicodeStringDestination* points to. If *UnicodeStringDestination* is non-**NULL**, this count never exceeds the value of *UnicodeStringMaxByteCount*. If *UnicodeStringDestination* is **NULL**, this count is the number of bytes that are required to contain the entire output string.

`UTF8StringSource`

A pointer to the UTF-8 source string.

`UTF8StringByteCount`

Specifies the number of bytes in the UTF-8 source string that the *UTF8StringSource* parameter points to.

Return Value

RtlUTF8ToUnicodeN returns STATUS_SUCCESS if the call is successful and all UTF-8 character codes in the input string were converted to the corresponding Unicode character codes in the output string. It returns STATUS_SOME_NOT_MAPPED if the call is successful but one or more input characters were invalid and were converted to the Unicode replacement character, U+FFFD. Possible error return values include the following error codes:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_TOO_SMALL	The *UnicodeStringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string.
STATUS_INVALID_PARAMETER	The *UnicodeStringDestination* and *UnicodeStringActualByteCount* parameters are both **NULL**.
STATUS_INVALID_PARAMETER_4	The *UTF8StringSource* parameter is **NULL**.

Remarks

The Unicode output string is null-terminated only if the UTF-8 input string is null-terminated.

The routine returns STATUS_BUFFER_TOO_SMALL if the *UnicodeStringMaxByteCount* parameter specifies a buffer size that is too small to contain the entire output string. In this case, the routine writes as many Unicode characters as will fit in the buffer, and the **UnicodeStringActualByteCount* value specifies the number of valid bytes that the routine has written to the buffer. The partial string that is contained in the output buffer might not include a terminating null character.

You can make an initial call to **RtlUTF8ToUnicodeN** to obtain the required output buffer size, and then call **RtlUTF8ToUnicodeN** again to obtain the Unicode output string. In the initial call, set *UnicodeStringDestination* = **NULL** and *UnicodeStringMaxByteCount* = 0, and the routine will write the required buffer size to **UnicodeStringActualByteCount*. Next, allocate a buffer of the required size and call **RtlUTF8ToUnicodeN** a second time to obtain the Unicode output string.

RtlUTF8ToUnicodeN supports Unicode surrogate pairs. However, a surrogate leading word value that is not followed by a trailing word value, or a trailing word value that is not preceded by a leading word value, is not recognized as a valid character and is replaced by the Unicode replacement character, U+FFFD.

RtlUTF8ToUnicodeN continues to convert the input string to an output string until it reaches the end of the source buffer or the end of the destination buffer, whichever occurs first. The routine converts any null characters in the input string to null characters in the output string. If the input string contains a terminating null character, but the null character is not located at the end of the source buffer, the routine continues past the terminating null character until it reaches the end of the available buffer space.

The RtlUnicodeToUTF8N routine converts a Unicode string to a UTF-8 string.

You can use the **RtlUTF8ToUnicode** and **RtlUnicodeToUTF8N** routines to perform a lossless conversion of valid text strings between the UTF-8 and Unicode formats. However, strings that have arbitrary data values are likely to violate the Unicode rules for encoding surrogate pairs, and any information that is contained in the invalid values in an input string is lost and cannot be recovered from the resulting output string.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntifs.h, Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeToUTF8N

RtlValidRelativeSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlValidRelativeSecurityDescriptor** routine checks the validity of a self-relative security descriptor.

Syntax

```
NTSYSAPI BOOLEAN RtlValidRelativeSecurityDescriptor(
  PSECURITY_DESCRIPTOR SecurityDescriptorInput,
  ULONG                SecurityDescriptorLength,
  SECURITY_INFORMATION RequiredInformation
);
```

Parameters

`SecurityDescriptorInput`

A pointer to the buffer that contains the security descriptor in self-relative format. The buffer must begin with a SECURITY_DESCRIPTOR structure, which is followed by the rest of the security descriptor data.

`SecurityDescriptorLength`

The size of the *SecurityDescriptorInput* structure.

`RequiredInformation`

A SECURITY_INFORMATION value that specifies the information that is required to be contained in the security descriptor.

Return Value

RtlValidRelativeSecurityDescriptor returns **TRUE** if the security descriptor is valid and includes the information that the *RequiredInformation* parameter specifies. Otherwise, this routine returns **FALSE**.

Remarks

To check the validity of a security descriptor in absolute format, use RtlValidSecurityDescriptor instead.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	"<= APC_LEVEL"

See Also

RtlValidSecurityDescriptor

SECURITY_DESCRIPTOR

SECURITY_INFORMATION

RtlValidSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlValidSecurityDescriptor** routine checks a given security descriptor's validity.

Syntax

```
NTSYSAPI BOOLEAN RtlValidSecurityDescriptor(
  PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Parameters

`SecurityDescriptor`

Pointer to the SECURITY_DESCRIPTOR to be checked.

Return Value

RtlValidSecurityDescriptor returns **TRUE** if the given descriptor is valid.

Remarks

The routine checks the validity of an absolute-format security descriptor. To check the validity of a self-relative security descriptor, use the RtlValidRelativeSecurityDescriptor routine instead.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCreateSecurityDescriptor

RtlLengthSecurityDescriptor

RtlSetDaclSecurityDescriptor

RtlValidRelativeSecurityDescriptor

SECURITY_DESCRIPTOR

RtlVerifyVersionInfo function

1/25/2019 • 3 minutes to read • Edit Online

The **RtlVerifyVersionInfo** routine compares a specified set of operating system version requirements to the corresponding attributes of the currently running version of the operating system.

Syntax

```
NTSYSAPI NTSTATUS RtlVerifyVersionInfo(
  PRTL_OSVERSIONINFOEXW VersionInfo,
  ULONG                 TypeMask,
  ULONGLONG             ConditionMask
);
```

Parameters

`VersionInfo`

Pointer to an RTL_OSVERSIONINFOEXW structure that specifies the operating system version requirements to compare to the corresponding attributes of the currently running version of the operating system.

`TypeMask`

Specifies which members of *VersionInfo* to compare with the corresponding attributes of the currently running version of the operating system. *TypeMask* is set to a logical OR of one or more of the following values.

VALUE	CORRESPONDING MEMBER
VER_BUILDNUMBER	**dwBuildNumber**
VER_MAJORVERSION	**dwMajorVersion**
VER_MINORVERSION	**dwMinorVersion**
VER_PLATFORMID	**dwPlatformId**
VER_SERVICEPACKMAJOR	**wServicePackMajor**
VER_SERVICEPACKMINOR	**wServicePackMinor**
VER_SUITENAME	**wSuiteMask**
VER_PRODUCT_TYPE	**wProductType**

`ConditionMask`

Specifies how to compare each **VersionInfo** member. To set the value of *ConditionMask*, a caller should use the **VER_SET_CONDITION** macro:

```
VER_SET_CONDITION (
        IN OUT  ULONGLONG  ConditionMask,
        IN ULONG  TypeBitMask,
        IN UCHAR  ComparisonType
        );
```

The value of *ConditionMask* is created in the following way:

- Initialize the value of *ConditionMask* to zero.
- Call **VER_SET_CONDITION** once for each *VersionInfo* member specified by *TypeMask*.
- Set the *TypeBitMask* and *ComparisonType* parameters for each call to **VER_SET_CONDITION** as follows:

TypeBitMask

Indicates the *VersionInfo* member for which the comparison type is set. *TypeBitMask* can be one of the following values.

VALUE	CORRESPONDING MEMBER
VER_BUILDNUMBER	**dwBuildNumber**
VER_MAJORVERSION	**dwMajorVersion**
VER_MINORVERSION	**dwMinorVersion**
VER_PLATFORMID	**dwPlatformId**
VER_SERVICEPACKMAJOR	**wServicePackMajor**
VER_SERVICEPACKMINOR	**wServicePackMinor**
VER_SUITENAME	**wSuiteMask**
VER_PRODUCT_TYPE	**wProductType**

ComparisonType

Specifies the comparison type that **RtlVerifyVersionInfo** uses to compare the **VersionInfo** member specified by *TypeBitMask* with the corresponding attribute of the currently running operating system.

For all values of *TypeBitMask* other than VER_SUITENAME, *ComparisonType* is set to one of the following values.

VALUE	MEANING
VER_EQUAL	The current value must be equal to the specified value.
VER_GREATER	The current value must be greater than the specified value.
VER_GREATER_EQUAL	The current value must be greater than or equal to the specified value.
VER_LESS	The current value must be less than the specified value.
VER_LESS_EQUAL	The current value must be less than or equal to the specified value.

If *TypeBitMask* is set to VER_SUITENAME, *ComparisonType* is set to of one the following values.

VALUE	MEANING
VER_AND	All product suites specified in the **wSuiteMask** member must be present in the current system.
VER_OR	At least one of the specified product suites must be present in the current system.

Return Value

RtlVerifyVersionInfo returns one of the following status values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The specified version matches the currently running version of the operating system.
STATUS_INVALID_PARAMETER	The input parameters are not valid.
STATUS_REVISION_MISMATCH	The specified version does not match the currently running version of the operating system.

Remarks

RtlVerifyVersionInfo enables a driver to easily verify the presence of a required set of operating system attributes. **RtlVerifyVersionInfo** is the kernel-mode equivalent of the user-mode **VerifyVersionInfo** function in the Windows SDK. See the example in the Windows SDK that shows how to verify the system version.

Typically, **RtlVerifyVersionInfo** returns STATUS_SUCCESS only if all comparisons succeed. However, the major version, minor version, and service pack version are tested in a sequential manner in the following way:

- If the major version exceeds the minimum required, then the minor version and service pack version are not tested. For example, if the current major version is 6.0, a test for a system greater than or equal to version 5.1 service pack 1 succeeds. The minor version and service pack version are not tested.

- If the minor version exceeds the minimum required, then the service pack version is not tested. For example, if the current major version is 5.2, a test for a system version greater than or equal to version 5.1 service pack 1 succeeds. The service pack version is not tested.
- If the major service pack version exceeds the minimum required, then the minor service pack version is not tested.

To verify a range of system versions, a driver can call **RtlVerifyVersionInfo** twice, once to verify a lower bound on the system version and once to verify an upper bound on the system version.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Ntddk.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RTL_OSVERSIONINFOEXW

RTL_OSVERSIONINFOW

RtlGetVersion

RtlWriteRegistryValue function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlWriteRegistryValue** routine writes caller-supplied data into the registry along the specified relative path at the given value name.

Syntax

```
NTSYSAPI NTSTATUS RtlWriteRegistryValue(
  ULONG  RelativeTo,
  PCWSTR Path,
  PCWSTR ValueName,
  ULONG  ValueType,
  PVOID  ValueData,
  ULONG  ValueLength
);
```

Parameters

`RelativeTo`

Specifies whether *Path* is an absolute registry path or is relative to a predefined path as one of the following.

VALUE	MEANING
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**. (For a system process, this is **\Users.Default**.)
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

`Path`

Pointer to either an absolute registry path or a path relative to the known location specified by the *RelativeTo*

parameter. If the RTL_REGISTRY_HANDLE flag is specified, this parameter is a registry handle for an already opened key to be used directly.

`ValueName`

Pointer to the name of a subkey or value entry to be written into the registry.

`ValueType`

Specifies a REG_*XXX* value that determines the type of the *ValueName* parameter. For a list of the possible values, see the *Type* parameter of ZwSetValueKey.

`ValueData`

Pointer to the name of a subkey or values for its value entries (or both) to be written into the registry.

`ValueLength`

Specifies the number of bytes of *ValueData* to be written into the registry.

Return Value

RtlWriteRegistryValue returns the status of the operation, either STATUS_SUCCESS or an error status.

Remarks

If the specified key does not exist, the routine attempts to create the key. For this attempt to succeed, the new key must be a direct subkey of the key that is referred to by the *Path* parameter, and the key that *Path* refers to must have been opened for KEY_CREATE_SUB_KEY access.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCheckRegistryKey

RtlCreateRegistryKey

RtlDeleteRegistryValue

RtlQueryRegistryValues

ZwCreateKey

ZwOpenKey

ZwSetValueKey

RtlxAnsiStringToUnicodeSize function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlxAnsiStringToUnicodeSize** routine returns the number of bytes that are required for a null-terminated Unicode string that is equivalent to a specified ANSI string.

Syntax

```
NTSYSAPI ULONG RtlxAnsiStringToUnicodeSize(
  PCANSI_STRING AnsiString
);
```

Parameters

`AnsiString`

Pointer to the ANSI string for which to compute the number of bytes that are required for an equivalent null-terminated Unicode string.

Return Value

RtlxAnsiStringToUnicodeSize returns the number of bytes that are required for an equivalent null-terminated Unicode string, if the ANSI string can be translated into an Unicode string by using the current system locale information. Otherwise, this routine returns zero.

Remarks

The ANSI string is interpreted for the current system locale.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAnsiStringToUnicodeSize

RtlxUnicodeStringToAnsiSize function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlxUnicodeStringToAnsiSize** routine returns the number of bytes required for a null-terminated ANSI string that is equivalent to a specified Unicode string.

Syntax

```
NTSYSAPI ULONG RtlxUnicodeStringToAnsiSize(
  PCUNICODE_STRING UnicodeString
);
```

Parameters

`UnicodeString`

Pointer to the Unicode string for which to compute the number of bytes required for an equivalent null-terminated ANSI string.

Return Value

If the Unicode string can be translated into an ANSI string using the current system locale information, **RtlxUnicodeStringToAnsiSize** returns the number of bytes required for an equivalent null-terminated ANSI string. Otherwise, it returns zero.

Remarks

The Unicode string is interpreted for the current system locale.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlUnicodeStringToAnsiSize

RtlZeroMemory macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlZeroMemory** routine fills a block of memory with zeros, given a pointer to the block and the length, in bytes, to be filled.

Syntax

```
void RtlZeroMemory(
   Destination,
   Length
);
```

Parameters

`Destination`

Datatype: void*. A pointer to the memory block to be filled with zeros.

`Length`

Datatype: size_t. The number of bytes to fill with zeros.

Return Value

None

Remarks

To zero out a memory buffer to erase security-sensitive data, use RtlSecureZeroMemory instead.

Callers of **RtlZeroMemory** can be running at any IRQL if the destination memory block is in nonpaged system memory. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Smclib.h, Minitape.h, Scsi.h, Storport.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (See Remarks section)

DDI compliance rules	BufAfterReqCompletedIntIoctlA, BufAfterReqCompletedIoctlA, BufAfterReqCompletedReadA, BufAfterReqCompletedWriteA

See Also

RtlFillMemory

RtlSecureZeroMemory

SeAccessCheck function

1/25/2019 • 2 minutes to read • Edit Online

The **SeAccessCheck** routine determines whether the requested access rights can be granted to an object protected by a security descriptor and an object owner.

Syntax

```
BOOLEAN SeAccessCheck(
  PSECURITY_DESCRIPTOR      SecurityDescriptor,
  PSECURITY_SUBJECT_CONTEXT SubjectSecurityContext,
  BOOLEAN                   SubjectContextLocked,
  ACCESS_MASK               DesiredAccess,
  ACCESS_MASK               PreviouslyGrantedAccess,
  PPRIVILEGE_SET            *Privileges,
  PGENERIC_MAPPING          GenericMapping,
  KPROCESSOR_MODE           AccessMode,
  PACCESS_MASK              GrantedAccess,
  PNTSTATUS                 AccessStatus
);
```

Parameters

`SecurityDescriptor`

Pointer to the SECURITY_DESCRIPTOR structure that describes the security descriptor protecting the object being accessed.

`SubjectSecurityContext`

Pointer to the SECURITY_SUBJECT_CONTEXT structure that specifies the subject's captured security context.

`SubjectContextLocked`

Indicates whether the user's subject context is locked, so that it does not have to be locked again.

`DesiredAccess`

Specifies the ACCESS_MASK bitmask for the access rights that the caller is attempting to acquire. If the caller sets the MAXIMUM_ALLOWED bit, the routine performs all DACL checks. However, the routine does not do any privilege checks, unless the caller specifically requests them by setting the ACCESS_SYSTEM_SECURITY or WRITE_OWNER bits.

`PreviouslyGrantedAccess`

Specifies the ACCESS_MASK bitmask of access rights already granted, such as access rights granted as a result of holding a privilege.

`Privileges`

Pointer to a caller-supplied variable to be set to the address of the PRIVILEGE_SET structure that will be used as part of the access validation, or this parameter can be **NULL**. The returned buffer, if any, must be released by the caller with SeFreePrivileges.

`GenericMapping`

Pointer to the GENERIC_MAPPING structure associated with this object type. This value specifies the specific access rights implied by each GENERIC_*XXX* access right.

`AccessMode`

Specifies the access mode to be used in the check, either **UserMode** or **KernelMode**.

`GrantedAccess`

Pointer to a returned access mask indicating the granted access. If the caller specifies MAXIMUM_ALLOWED, and the DACL in *SecurityDescriptor* is **NULL**, then the routine returns GENERIC_ALL plus any additional access the caller explicitly requests.

`AccessStatus`

Pointer to the status value indicating why access was denied.

Return Value

If access is allowed, **SeAccessCheck** returns **TRUE**.

Remarks

SeAccessCheck might perform privilege tests for **SeTakeOwnershipPrivilege** and **SeSecurityPrivilege**, depending on the accesses being requested. It might perform additional privilege testing in future releases of the operating system.

This routine also might check whether the caller is the owner of the object in order to grant WRITE_DAC or READ_CONTROL access.

If this routine returns **FALSE**, the caller should use the returned *AccessStatus* as its return value. That is, the caller should avoid hardcoding a return value of STATUS_ACCESS_DENIED or any other specific STATUS_*XXX* value.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

GENERIC_MAPPING

IoGetFileObjectGenericMapping

PRIVILEGE_SET

SECURITY_DESCRIPTOR

SECURITY_SUBJECT_CONTEXT

SeFreePrivileges

SeValidSecurityDescriptor

SeAssignSecurity function

1/25/2019 • 2 minutes to read • Edit Online

The **SeAssignSecurity** routine builds a self-relative security descriptor for a new object, given the security descriptor of its parent directory and any originally requested security for the object.

Syntax

```
NTSTATUS SeAssignSecurity(
  PSECURITY_DESCRIPTOR      ParentDescriptor,
  PSECURITY_DESCRIPTOR      ExplicitDescriptor,
  PSECURITY_DESCRIPTOR      *NewDescriptor,
  BOOLEAN                   IsDirectoryObject,
  PSECURITY_SUBJECT_CONTEXT SubjectContext,
  PGENERIC_MAPPING          GenericMapping,
  POOL_TYPE                 PoolType
);
```

Parameters

`ParentDescriptor`

Pointer to a buffer containing the SECURITY_DESCRIPTOR for the parent directory, if any, containing the new object being created. *ParentDescriptor* can be **NULL**, or have a **NULL** system access control list (SACL) or a **NULL** discretionary access control list (DACL).

`ExplicitDescriptor`

Pointer to a buffer containing the SECURITY_DESCRIPTOR specified by the user that is applied to the new object. *ExplicitDescriptor* can be **NULL**, or have a **NULL** SACL or a **NULL** DACL.

`NewDescriptor`

Receives a pointer to the returned SECURITY_DESCRIPTOR. **SeAssignSecurity** allocates the buffer from the paged memory pool.

`IsDirectoryObject`

Specifies whether the new object is a directory object. **TRUE** indicates the object contains other objects.

`SubjectContext`

Pointer to a buffer containing the security context of the subject creating the object. This is used to retrieve default security information for the new object, such as the default owner, the primary group, and discretionary access control.

`GenericMapping`

Pointer to the GENERIC_MAPPING structure that describes the mapping from each generic right to the implied nongeneric rights.

`PoolType`

This parameter is unused. The buffer to hold the new security descriptor is always allocated from paged pool.

Return Value

SeAssignSecurity can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The assignment was successful.
STATUS_INVALID_OWNER	The SID provided for the owner of the target security descriptor is not one the caller is authorized to assign as the owner of an object.
STATUS_PRIVILEGE_NOT_HELD	The caller does not have the privilege (**SeSecurityPrivilege**) necessary to explicitly assign the specified system ACL.

Remarks

The final security descriptor returned to the caller may contain a mix of information, some explicitly provided from the new object's parent.

SeAssignSecurity assumes privilege checking has not been performed. This routine performs privilege checking.

The assignment of system and discretionary ACLs is governed by the logic illustrated in the following table:

	EXPLICIT (NONDEFAULT) ACL SPECIFIED	EXPLICIT DEFAULT ACL SPECIFIED	NO ACL SPECIFIED
Inheritable ACL from parent	Assign specified ACL	Assign inherited ACL	Assign inherited ACL
No inheritable ACL from parent	Assign specified ACL	Assign default ACL	Assign no ACL

An explicitly specified ACL, whether a default ACL or not, can be empty or null. The caller must be a kernel-mode client or be appropriately privileged to explicitly assign a default or nondefault system ACL.

The assignment of the new object's owner and group is governed by the following logic:

- If the passed security descriptor includes an owner, it is assigned as the new object's owner. Otherwise, the caller's token is considered to determine the owner. Within the token, the default owner, if any, is assigned. Otherwise, the caller's user ID is assigned.

- If the passed security descriptor includes a group, it is assigned as the new object's group. Otherwise, the caller's token is considered to determine the group. Within the token, the default group, if any, is assigned. Otherwise, the caller's primary group ID is assigned.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

GENERIC_MAPPING

IoGetFileObjectGenericMapping

SECURITY_DESCRIPTOR

SeDeassignSecurity

SeAssignSecurityEx function

1/25/2019 • 4 minutes to read • Edit Online

The **SeAssignSecurityEx** routine builds a self-relative security descriptor for a new object given the following optional parameters: a security descriptor of the object's parent directory, an explicit security descriptor for the object, and the object type.

Syntax

```
NTSTATUS SeAssignSecurityEx(
  PSECURITY_DESCRIPTOR      ParentDescriptor,
  PSECURITY_DESCRIPTOR      ExplicitDescriptor,
  PSECURITY_DESCRIPTOR      *NewDescriptor,
  GUID                      *ObjectType,
  BOOLEAN                   IsDirectoryObject,
  ULONG                     AutoInheritFlags,
  PSECURITY_SUBJECT_CONTEXT SubjectContext,
  PGENERIC_MAPPING          GenericMapping,
  POOL_TYPE                 PoolType
);
```

Parameters

`ParentDescriptor`

Pointer to the SECURITY_DESCRIPTOR of the parent object that contains the new object being created. *ParentDescriptor* can be **NULL**, or have a **NULL** system access control list (SACL) or a **NULL** discretionary access control list (DACL).

`ExplicitDescriptor`

Pointer to an explicit SECURITY_DESCRIPTOR that is applied to the new object. *ExplicitDescriptor* can be **NULL**, or have a **NULL** SACL or a **NULL** DACL.

`NewDescriptor`

Receives a pointer to the returned SECURITY_DESCRIPTOR. **SeAssignSecurityEx** allocates the buffer from the paged memory pool.

`ObjectType`

Pointer to a GUID for the type of object being created. If the object does not have a GUID, *ObjectType* must be set to **NULL**.

`IsDirectoryObject`

Specifies whether the new object is a directory object. If *IsDirectoryObject* is set to **TRUE**, the new object is a directory object, otherwise the new object is not a directory object.

`AutoInheritFlags`

Specifies the type of automatic inheritance that is applied to access control entries (ACE) in the access control lists (ACL) specified by *ParentDescriptor*. *AutoInheritFlags* also controls privilege checking, owner checking, and setting a default owner and group for *NewDescriptor*. *AutoInheritFlags* must be set to a logical OR of one or more of the

following values:

VALUE	MEANING
SEF_DACL_AUTO_INHERIT	ACEs in the DACL of *ParentDescriptor* are inherited by *NewDescriptor*, in addition to explicit ACEs specified by *ExplicitDescriptor*.
SEF_SACL_AUTO_INHERIT	ACEs in the SACL of *ParentDescriptor* are inherited by *NewDescriptor*, in addition to explicit ACEs specified by *ExplicitDescriptor*.
SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT	*ExplicitDescriptor* is the default descriptor for the object type specified by *ObjectType*. *ExplicitDescriptor* is not used if ACEs are inherited from *ParentDescriptor*.
SEF_AVOID_PRIVILEGE_CHECK	Privilege checking is not done. This flag is useful with automatic inheritance because it avoids privilege checking on each child that needs to be updated.
SEF_AVOID_OWNER_CHECK	Owner checking is not done.
SEF_DEFAULT_OWNER_FROM_PARENT	If an owner is specified by *ExplicitDescriptor*, this flag is not used, and the owner of *NewDescriptor* is set to the owner specified by *ExplictDescriptor*. If an owner is not specified by *ExplicitDescriptor*, this flag is used in the following way: If the flag is set, the owner of *NewDescriptor* is set to the owner of *ParentDescriptor*. Otherwise, the owner of *NewDescriptor* is set to the owner specified by the *SubjectContext*.
SEF_DEFAULT_GROUP_FROM_PARENT	If a group is specified by *ExplicitDescriptor*, this flag is not used, and the group of *NewDescriptor* is set to the group specified by *ExplictDescriptor*. If a group is not specified by *ExplicitDescriptor*, this flag is used in the following way: If the flag is set, the group of *NewDescriptor* is set to the group of *ParentDescriptor*. Otherwise, the group of *NewDescriptor* is set to the group specified by the *SubjectContext*.

The assignment of system and discretionary ACLs is described in the following table:

	NONDEFAULT EXPLICIT DESCRIPTOR(1)	DEFAULT EXPLICIT DESCRIPTOR(2)	NULL EXPLICIT DESCRIPTOR
ACL is inherited from parent descriptor(3).	Assign both inherited and explicit ACLs(5)(6).	Assign inherited ACL.	Assign inherited ACL.
ACL is not inherited from parent descriptor(4).	Assign nondefault ACL.	Assign default ACL.	Assign no ACL.

Assignment Notes

1. The SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT flag is not specified.
2. The SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT flag is specified.
3. The auto inherit flag for an ACL is specified (SEF_DACL_AUTO_INHERIT or SEF_SACL_AUTO_INHERIT).
4. The automatic inherit flag for an ACL is not specified.
5. ACEs with the INHERITED_ACE bit set in their **AceFlags** member are not copied to the assigned security descriptor.
6. ACEs that are inherited from the parent descriptor are appended after the ACEs specified by the explicit

descriptor.

`SubjectContext`

Pointer to a security context of the subject that is creating the object. *SubjectContext* is used to retrieve default security information for the new object, including the default owner, the primary group, and discretionary access control.

`GenericMapping`

Pointer to an array of access mask values that specify the mapping between each generic rights to object-specific rights.

`PoolType`

This parameter is unused. The buffer to hold the new security descriptor is always allocated from paged pool.

Return Value

SeAssignSecurityEx returns one of the following values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The assignment was successful.
STATUS_INVALID_OWNER	The SID provided as the owner of the new security descriptor is not a SID that the caller is authorized to assign as the owner of an object.
STATUS_PRIVILEGE_NOT_HELD	The caller does not have the privilege (**SeSecurityPrivilege**) necessary to explicitly assign the specified SACL.

Remarks

SeAssignSecurityEx extends the basic operation of **SeAssignSecurity** in the following ways:

- *ObjectType* optionally specifies an object type. Object-specific inheritance is controlled by the following members of an object-specific ACE: **Flags**, **InheritedObjectType**, and **Header.AceFlags**.
- *AutoInheritFlags*specifies the type of automatic inheritance of ACEs that is used. AutoInheritFlags also controls privilege checking, owner checking, and setting a default owner and group for *NewDescriptor*.

For more information about security and access control, see the documentation on these topics in the Microsoft Windows SDK.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	HwStorPortProhibitedDDIs

See Also

GENERIC_MAPPING

SECURITY_DESCRIPTOR

SeAssignSecurity

SeDeassignSecurity

SeDeassignSecurity function

1/25/2019 • 2 minutes to read • Edit Online

The
SeDeassignSecurity routine deallocates the memory associated with a security descriptor that was assigned using **SeAssignSecurity**.

Syntax

```
NTSTATUS SeDeassignSecurity(
  PSECURITY_DESCRIPTOR *SecurityDescriptor
);
```

Parameters

`SecurityDescriptor`

Pointer to the buffered SECURITY_DESCRIPTOR being released.

Return Value

If the deallocation succeeds, **SeDeassignSecurity** returns STATUS_SUCCESS.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

SECURITY_DESCRIPTOR

SeAssignSecurity

SET_D3COLD_SUPPORT callback function

1/25/2019 • 4 minutes to read • Edit Online

The *SetD3ColdSupport* routine enables or disables transitions to the D3cold device power state.

Syntax

```
SET_D3COLD_SUPPORT SetD3coldSupport;

void SetD3coldSupport(
  PVOID Context,
  BOOLEAN D3ColdSupport
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller sets this parameter to the value of the **Context** member of the D3COLD_SUPPORT_INTERFACE structure for the interface.

`D3ColdSupport`

Whether to allow transitions from D3hot to D3cold. Set to **TRUE** to enable these transitions. Otherwise, set to **FALSE.** For more information, see Remarks.

Return Value

None.

Remarks

The driver that is the power policy owner (PPO) for a device can call this routine to enable or disable transitions to the D3cold substate that might occur when the computer is in the S0 (working) system power state and is not preparing to exit S0.

If a device must be able to signal a wake event from any low-power Dx state that it enters, the driver for this device should not enable transitions to the D3cold substate unless the device can signal a wake event from this substate. Otherwise, after entering D3cold, the device will be unavailable until the computer restarts or wakes from a sleeping state.

The driver for the device can call the GetIdleWakeInfo routine to determine whether the device can signal a wake event from D3cold.

If necessary, the driver for a device can make a series of *SetD3ColdSupport* calls to alternately enable and disable D3cold transitions in response to dynamically changing conditions.

For example, a graphics device might have a large hardware context to restore after a transition from D3cold to D0. The time required to restore this context may or may not be acceptable depending on the set of tasks that the user is currently performing. In addition, the restore time might vary with this task mix. The driver might need to dynamically enable and disable transitions to D3cold as the task mix changes.

A device can enter the D3cold substate only from the D3hot substate. If the driver calls *SetD3ColdSupport* to enable transitions from D3hot to D3cold, the device may or may not enter D3cold after it enters D3hot. In response to a wake event, a device might enter D0 directly from D3hot without ever passing through D3cold.

When a device moves from D3hot to D3cold, it probably does so because the power source that it shares with some number of other devices was turned off. Some time after these devices enter D3cold, the driver for one of the devices might request a transition to D0. In response to this request, the parent bus driver or ACPI filter driver turns on the power source, and all the devices that share the power source enter their default, power-on states.

The only device driver that expects this power state change is the driver that requested the change. The drivers for the other devices must receive notification of this change so that they can properly initialize their devices to operate in D0. Only a driver that can receive this notification should enable its device to enter D3cold. Otherwise, the driver will not know when the device enters D0.

Starting with Windows 8, a WDM driver can register its device with the power framework (PoFx) and be notified by the DevicePowerRequiredCallback routine when the device enters D0. Similarly, a KMDF driver can register its device with PoFx and be notified by the EvtDeviceD0Entry event callback.

A driver that does not register its device with PoFx can still be notified of a transition to D0 if the device is armed for wake. When the bus drivers turn on the power to the device, they complete the driver's IRP_MN_WAIT_WAKE request. In response, the driver initializes its device to operate in D0.

After a power source shared by several devices turns on, the device whose driver requested the transition to D0 might be the only one of these devices that has work to do. The other devices are likely to be idle, in which case their drivers, after some time, will move these devices to D3hot.

If the driver calls *SetD3ColdSupport* to disable transitions to D3cold, D0 is the only device power state that the device can enter from D3hot.

If the driver calls *SetD3ColdSupport* to enable transitions to D3cold, these transitions might be disabled for other reasons. The driver can call the GetD3ColdCapability routine to determine whether the device is capable of entering the D3cold substate. However, a driver might not need this information. If a device is not capable of entering D3cold, calls to *SetD3ColdSupport* have no effect, but are harmless.

The *SetD3ColdSupport* routine affects only D3hot-to-D3cold transitions that might occur when the computer is in (and is not preparing to exit) the S0 state. This routine has no effect on the case in which the computer is preparing to exit S0 and enter a low-power system state. A device in the D3hot substate might always enter the D3cold substate just before the computer enters a system low-power state.

By default, before the first call to the *SetD3ColdSupport* routine, D3hot-to-D3cold transitions are disabled. To change this default so that D3hot-to-D3cold transitions are enabled before the first *SetD3ColdSupport* call, the driver package for the device can include the following two lines in the **DDInstall.HW** section of the INF file that installs the driver:

```
Include = machine.inf
Needs = PciD3ColdSupported
```

Windows 8 is the first version of Windows to support devices that can enter and exit the D3cold substate while the computer remains in the S0 state. For more information, see Device Low-Power States.

Requirements

Minimum supported client	Available starting with Windows 8.
Target Platform	Desktop
Header	wdm.h (include Wdm.h)
IRQL	PASSIVE_LEVEL

See Also

D3COLD_SUPPORT_INTERFACE

DevicePowerRequiredCallback

EvtDeviceD0Entry

GetD3ColdCapability

IRP_MN_WAIT_WAKE

SeValidSecurityDescriptor function

1/25/2019 • 2 minutes to read • Edit Online

The **SeValidSecurityDescriptor** routine returns whether a given security descriptor is structurally valid.

Syntax

```
BOOLEAN SeValidSecurityDescriptor(
  ULONG                Length,
  PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Parameters

`Length`

Specifies the size in bytes of the given security descriptor.

`SecurityDescriptor`

Pointer to the self-relative SECURITY_DESCRIPTOR, which must be buffered somewhere in system space.

Return Value

SeValidSecurityDescriptor returns **TRUE** if the buffered security descriptor is structurally valid.

Remarks

SeValidSecurityDescriptor does not enforce policy. It simply checks that the given security descriptor data is formatted correctly. In particular, it checks the revision information, self relativity, owner, alignment, and, if available, SID, group, DACL, ACL, and/or SACL do not overflow the given *Length*. Consequently, callers of **SeValidSecurityDescriptor** cannot assume that a returned **TRUE** implies that the given security descriptor necessarily has valid contents.

If **SeValidSecurityDescriptor** returns **TRUE**, the given security descriptor can be passed on to another kernel-mode component because it is structurally valid. Otherwise, passing a structurally invalid security descriptor to be manipulated by another kernel-mode component can cause undefined results or even a system bug check.

To validate a security descriptor that was passed in from user mode, call **RtlValidSecurityDescriptor** rather than **SeValidSecurityDescriptor**.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

RtlValidSecurityDescriptor

SECURITY_DESCRIPTOR

TmCommitComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **TmCommitComplete** routine notifies KTM that the calling resource manager has finished committing a transaction.

Syntax

```
NTSTATUS TmCommitComplete(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmCommitComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_REQUESTED	The transaction or its enlistment is not in the correct state.

The routine might return other NTSTATUS values.

Remarks

The **TmCommitComplete** routine is a pointer-based version of the ZwCommitComplete routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCommitComplete

ZwCreateEnlistment

ZwOpenEnlistment

TmCommitEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmCommitEnlistment** routine initiates the commit operation for a specified enlistment's transaction.

Syntax

```
NTSTATUS TmCommitEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmCommitEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_COMMIT_COMPLETE notifications.
STATUS_TRANSACTION_NOT_VALID	The enlistment's transaction is not in a state that enables it to be committed.
STATUS_NOT_ACTIVE	The commit operation for this transaction has already been started.
STATUS_TRANSACTION_ALREADY_ABORTED	The transaction cannot be committed because it has been rolled back.

The routine might return other NTSTATUS values.

Remarks

The **TmCommitEnlistment** routine is a pointer-based version of the ZwCommitEnlistment routine.

Only superior transaction managers can call **TmCommitEnlistment**.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCommitEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

TmCommitTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **TmCommitTransaction** routine initiates a commit operation for a specified transaction.

Syntax

```
NTSTATUS TmCommitTransaction(
  PKTRANSACTION Transaction,
  BOOLEAN       Wait
);
```

Parameters

`Transaction`

A pointer to a transaction object. To obtain this pointer, your TPS component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransaction or ZwOpenTransaction provided.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is **TRUE**, the call returns after the commit operation is complete.

Return Value

TmCommitTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_SUPERIOR_EXISTS	The caller cannot commit the transaction because a superior transaction manager exists.

STATUS_TRANSACTION_ABORTED	The transaction cannot be committed because it has been rolled back.
STATUS_TRANSACTION_COMMITTED	The transaction is already committed.
STATUS_TRANSACTION_NOT_VALID	The commit operation for this transaction has already been started.
STATUS_PENDING	Commit notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

The **TmCommitTransaction** routine is a pointer-based version of the ZwCommitTransaction routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ZwCommitTransaction

ZwCreateTransaction

ZwOpenTransaction

TmCreateEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmCreateEnlistment** routine creates a new enlistment object for a transaction.

Syntax

```
NTSTATUS TmCreateEnlistment(
  PHANDLE            EnlistmentHandle,
  KPROCESSOR_MODE    PreviousMode,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PRKRESOURCEMANAGER ResourceManager,
  PKTRANSACTION      Transaction,
  ULONG              CreateOptions,
  NOTIFICATION_MASK  NotificationMask,
  PVOID              EnlistmentKey
);
```

Parameters

`EnlistmentHandle`

A pointer to a caller-allocated variable that receives a handle to the new enlistment object if the call to **TmCreateEnlistment** succeeds.

`PreviousMode`

The processor mode of the process that will use the enlistment handle to access the enlistment object. This value must be either **UserMode** or **KernelMode**.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the enlistment object. For more information about this parameter, see the description of the *DesiredAccess* parameter for ZwCreateEnlistment.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`ResourceManager`

A pointer to a resource manager object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateResourceManager or ZwOpenResourceManager provided.

`Transaction`

A pointer to a transaction object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransaction or ZwOpenTransaction provided. KTM adds this transaction to the list of transactions that the calling resource manager is handling.

`CreateOptions`

Enlistment option flags. The following table contains the only available flag.

CREATEOPTIONS FLAG	MEANING
ENLISTMENT_SUPERIOR	The caller is enlisting as a superior transaction manager for the specified transaction.

This parameter is optional and can be zero.

`NotificationMask`

A bitwise OR of the TRANSACTION_NOTIFY_*XXX* values that are defined in Ktmtypes.h. This mask value specifies the types of transaction notifications that KTM sends to the caller.

`EnlistmentKey`

A pointer to caller-defined information that uniquely identifies the enlistment. The resource manager receives this pointer when it calls ZwGetNotificationResourceManager or when KTM calls the ResourceManagerNotification callback routine. The resource manager can maintain a reference count for this key by calling TmReferenceEnlistmentKey and TmDereferenceEnlistmentKey. This parameter is optional and can be **NULL**.

Return Value

TmCreateEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of the *CreateOptions* or *NotificationMask* parameter is invalid, or KTM could not find the transaction that the *Transaction* parameter specifies.
STATUS_INSUFFICIENT_RESOURCES	A memory allocation failed.

STATUS_TRANSACTION_MANAGER_NOT_ONLINE	The enlistment failed because KTM or the resource manager is not in an operational state.
STATUS_TRANSACTION_NOT_ACTIVE	The enlistment failed because the transaction that the *Transaction* parameter specifies is not active.
STATUS_TRANSACTION_SUPERIOR_EXISTS	The caller tried to register as a superior transaction manager but a superior enlistment already exists.
STATUS_TM_VOLATILE	The caller is trying to register as a superior transaction manager, but the caller's resource manager object is volatile while the associated transaction manager object is not volatile.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

The **TmCreateEnlistment** routine is a pointer-based version of the ZwCreateEnlistment routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

ObReferenceObjectByHandle

ResourceManagerNotification

ZwCreateEnlistment

ZwCreateResourceManager

ZwCreateTransaction

ZwGetNotificationResourceManager

ZwOpenResourceManager

ZwOpenTransaction

TmDereferenceEnlistmentKey function

1/25/2019 • 2 minutes to read • Edit Online

The **TmDereferenceEnlistmentKey** routine decrements the reference count for the key of a specified enlistment object.

Syntax

```
NTSTATUS TmDereferenceEnlistmentKey(
  PKENLISTMENT Enlistment,
  PBOOLEAN     LastReference
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`LastReference`

A pointer to a BOOLEAN-typed variable. This variable receives **TRUE** if the reference count is zero after **TmDereferenceEnlistmentKey** decrements it. Otherwise, the variable receives **FALSE**.

Return Value

TmDereferenceEnlistmentKey returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_UNSUCCESSFUL	The reference count of the specified enlistment object is zero and cannot be decremented.

The routine might return other NTSTATUS values.

Remarks

The TmReferenceEnlistmentKey routine increments the reference count for an enlistment object's key, and the **TmDereferenceEnlistmentKey** routine decrements the count.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

TmReferenceEnlistmentKey

ZwCreateEnlistment

ZwOpenEnlistment

TmEnableCallbacks function

1/25/2019 • 2 minutes to read • Edit Online

The **TmEnableCallbacks** routine enables a callback routine that receives transaction notifications.

Syntax

```
NTSTATUS TmEnableCallbacks(
  PKRESOURCEMANAGER   ResourceManager,
  PTM_RM_NOTIFICATION CallbackRoutine,
  PVOID               RMKey
);
```

Parameters

`ResourceManager`

A pointer to a resource manager object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateResourceManager or ZwOpenResourceManager provided.

`CallbackRoutine`

A pointer to a ResourceManagerNotification routine.

`RMKey`

A caller-defined context value that uniquely identifies the resource manager. The caller's *ResourceManagerNotification* routine receives this value as input.

Return Value

TmEnableCallbacks returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STATUS_UNSUCCESSFUL	The *CallbackRoutine* pointer is **NULL**.

The routine might return other NTSTATUS values.

Remarks

Your resource manager can call **TmEnableCallbacks** to enable a ResourceManagerNotification callback routine. Use **TmEnableCallbacks** to enable asynchronous notifications. Use ZwGetNotificationResourceManager to

obtain notifications synchronously.

Resource managers can register one callback routine for each resource manager object.

For more information about the **TmEnableCallbacks** routine, see Creating a Resource Manager.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

ZwCreateResourceManager

ZwGetNotificationResourceManager

ZwOpenResourceManager

TmGetTransactionId function

1/25/2019 • 2 minutes to read • Edit Online

The **TmGetTransactionId** routine retrieves a transaction object's unit of work (UOW) identifier.

Syntax

```
void TmGetTransactionId(
  PKTRANSACTION Transaction,
  PUOW          TransactionId
);
```

Parameters

`Transaction`

A pointer to a transaction object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransaction or ZwOpenTransaction provided.

`TransactionId`

A pointer to a location that receives the transaction object's UOW identifier.

Return Value

None

Remarks

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ObReferenceObjectByHandle

ZwCreateTransaction

ZwOpenTransaction

ZwQueryInformationTransaction

TmInitializeTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **TmInitializeTransactionManager** routine initializes a transaction manager object.

Syntax

```
NTSTATUS TmInitializeTransactionManager(
  PRKTM            TransactionManager,
  PCUNICODE_STRING LogFileName,
  PGUID            TmId,
  ULONG            CreateOptions
);
```

Parameters

`TransactionManager`

A pointer to the transaction manager object to initialize.

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of a CLFS log file stream associated with the transaction manager object.

`TmId`

Specifies a pointer to a GUID that identifies the name of the transaction manager object to initialize.

`CreateOptions`

Optional object creation flags. The following table contains the available flags.

OPTION FLAG	MEANING
TRANSACTION_MANAGER_VOLATILE	The transaction manager object will be volatile. Therefore, it will not use a log file.
TRANSACTION_MANAGER_COMMIT_DEFAULT	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_VOLUME	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_HIVES	For internal use only.
TRANSACTION_MANAGER_COMMIT_LOWEST	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_RECOVERY	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_PROGRESS	For internal use only.

Return Value

The **TmInitializeTransactionManager** routine returns STATUS_SUCCESS if the operation succeeds.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h

TmIsTransactionActive function

1/25/2019 • 2 minutes to read • Edit Online

The **TmIsTransactionActive** routine indicates whether a specified transaction is in its active state.

Syntax

```
BOOLEAN TmIsTransactionActive(
  PKTRANSACTION Transaction
);
```

Parameters

`Transaction`

A pointer to a transaction object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransaction or ZwOpenTransaction provided.

Return Value

TmIsTransactionActive returns **TRUE** if the specified transaction is in its active state. Otherwise, the routine returns **FALSE**.

Remarks

When a transaction is created, it enters its active state and remains in that state until the pre-prepare/prepare/commit sequence begins.

For more information about **TmIsTransactionActive**, and for information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

ObReferenceObjectByHandle

ZwCreateTransaction

ZwOpenTransaction

TmPrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **TmPrepareComplete** routine notifies KTM that the calling resource manager has finished preparing a transaction's data.

Syntax

```
NTSTATUS TmPrepareComplete(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmPrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOTIFY_REQUESTED	The transaction or its enlistment is not in the correct state. For example, KTM did not send a TRANSACTION_NOTIFY_PREPREPARE notification.

The routine might return other NTSTATUS values.

Remarks

The **TmPrepareComplete** routine is a pointer-based version of the ZwPrepareComplete routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

TmPrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmPrepareEnlistment** routine initiates the prepare operation for a specified enlistment's transaction.

Syntax

```
NTSTATUS TmPrepareEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmPrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STA TUS _TR AN SAC TIO N_R ESP ON SE_ NO T_E NLI STE D	The caller did not register to receive TRANSACTION_NOTIFY_PREPARE_COMPLETE notifications.
STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment's transaction is not in a state that allows it to enter the prepare phase.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

The **TmPrepareEnlistment** routine is a pointer-based version of the ZwPrepareEnlistment routine.

Only superior transaction managers can call **TmPrepareEnlistment**.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareEnlistment

TmPrePrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **TmPrePrepareComplete** routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

Syntax

```
NTSTATUS TmPrePrepareComplete(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmPrePrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOTIFY_REQUESTED	The transaction or its enlistment is not in the correct state. For example, KTM did not send a TRANSACTION_NOTIFY_PREPARE notification.

The routine might return other NTSTATUS values.

Remarks

The **TmPrePrepareComplete** routine is a pointer-based version of the ZwPrePrepareComplete routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrePrepareComplete

TmPrePrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmPrePrepareEnlistment** routine initiates the pre-prepare operation for a specified enlistment's transaction.

Syntax

```
NTSTATUS TmPrePrepareEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmPrePrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STA TUS _TR AN SAC TIO N_R ESP ON SE_ NO T_E NLI STE D	The caller did not register to receive TRANSACTION_NOTIFY_PREPREPARE_COMPLETE notifications.
STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The transaction's state does not allow it to enter the pre-prepare phase.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

The **TmPrePrepareEnlistment** routine is a pointer-based version of the ZwPrePrepareEnlistment routine.

Only superior transaction managers can call **TmPrePrepareEnlistment**.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrePrepareEnlistment

TmReadOnlyEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmReadOnlyEnlistment** routine sets a specified enlistment to be read-only.

Syntax

```
NTSTATUS TmReadOnlyEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmReadOnlyEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_REQUESTED	The transaction or its enlistment is not in the correct state. For example, an enlistment cannot be read-only if the ENLISTMENT_SUPERIOR flag was specified to ZwCreateEnlistment.

The routine might return other NTSTATUS values.

Remarks

The **TmReadOnlyEnlistment** routine is a pointer-based version of the ZwReadOnlyEnlistment routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwReadOnlyEnlistment

TmRecoverEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRecoverEnlistment** routine initiates a recovery operation for the transaction that is associated with a specified enlistment.

Syntax

```
NTSTATUS TmRecoverEnlistment(
  PKENLISTMENT Enlistment,
  PVOID        EnlistmentKey
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`EnlistmentKey`

A pointer to the enlistment key value that the resource manager previously specified as the *EnlistmentKey* parameter to ZwCreateEnlistment or TmCreateEnlistment. This parameter is optional and can be **NULL** if the resource manager did not provide an enlistment key when it called **ZwCreateEnlistment** or **TmCreateEnlistment**.

Return Value

TmRecoverEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STA TUS _OB JEC T_T YPE _MI SM ATC H	The specified handle is not a handle to an enlistment object.

STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_TRANSACTION_NOT_ACTIVE	The transaction or its enlistment is not in the correct state.
STATUS_NOTIFICATION_QUEUE_FULL	KTM has added a commit, rollback, or in-doubt notification to the resource manager's notification queue, and the resource manager is not using a ResourceManagerNotification callback routine.

The routine might return other NTSTATUS values.

Remarks

The **TmRecoverEnlistment** routine is a pointer-based version of the ZwRecoverEnlistment routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

For more information about recovery operations, see Handling Recovery Operations.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TRANSACTION_NOTIFICATION

TmCreateEnlistment

ZwCreateEnlistment

ZwGetNotificationResourceManager

ZwOpenEnlistment

ZwRecoverEnlistment

TmRecoverResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRecoverResourceManager** routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.

Syntax

```
NTSTATUS TmRecoverResourceManager(
  PKRESOURCEMANAGER ResourceManager
);
```

Parameters

`ResourceManager`

A pointer to a resource manager object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateResourceManager or ZwOpenResourceManager provided.

Return Value

TmRecoverResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_ROOT	The transaction manager that is associated with the specified resource manager is not online.

The routine might return other NTSTATUS values.

Remarks

The **TmRecoverResourceManager** routine is a pointer-based version of the ZwRecoverResourceManager routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ZwCreateResourceManager

ZwOpenResourceManager

ZwRecoverResourceManager

TmRecoverTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRecoverTransactionManager** routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.

Syntax

```
NTSTATUS TmRecoverTransactionManager(
  PKTM           Tm,
  PLARGE_INTEGER TargetVirtualClock
);
```

Parameters

`Tm`

A pointer to a transaction manager object. To obtain this pointer, your TPS component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager provided.

`TargetVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**. For more information about this parameter, see the following Remarks section. For Windows Vista, this parameter must be **NULL**.

Return Value

TmRecoverTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_NOT_IMPLEMENTED	For Windows Vista, the caller specified a non-**NULL** value for the *TargetVirtualClock* parameter.
STATUS_TM_VOLATILE	The specified transaction manager object does not have a log file, so recovery is not available.

STA TUS _UN SUC CES SFU L	The specified transaction manager object is in an unexpected state.

The routine might return other NTSTATUS values.

Remarks

The **TmRecoverTransactionManager** routine is a pointer-based version of the ZwRecoverTransactionManager routine.

Calling **TmRecoverTransactionManager** with a *TargetVirtualClock* parameter value of **NULL** is functionally equivalent to calling **ZwRecoverTransactionManager**. Calling **TmRecoverTransactionManager** with a non-**NULL** *TargetVirtualClock* parameter value is functionally equivalent to calling ZwRollforwardTransactionManager.

For more information about recovery operations, see Handling Recovery Operations.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ZwRecoverTransactionManager

ZwRollforwardTransactionManager

TmReferenceEnlistmentKey function

1/25/2019 • 2 minutes to read • Edit Online

The **TmReferenceEnlistmentKey** routine increments the reference count for the key of a specified enlistment object and retrieves the key.

Syntax

```
NTSTATUS TmReferenceEnlistmentKey(
  PKENLISTMENT Enlistment,
  PVOID        *Key
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`Key`

A pointer to a variable that receives the enlistment object's enlistment key. The caller assigns an enlistment key when it calls ZwCreateEnlistment or TmCreateEnlistment.

Return Value

TmReferenceEnlistmentKey returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *Key* parameter's value is **NULL**.

STATUS_UNSUCCESSFUL	The reference count of the specified enlistment object has been decremented to zero, so the reference count cannot be incremented.
STATUS_INSUFFICIENT_RESOURCES	The reference count is at its maximum value (0xFFFFFFFF) and cannot be incremented.

The routine might return other NTSTATUS values.

Remarks

The **TmReferenceEnlistmentKey** routine increments the reference count for an enlistment object's key value, and the TmDereferenceEnlistmentKey routine decrements the count.

If a resource manager has defined a key value for an enlistment, the resource manager receives the key value when it calls ZwGetNotificationResourceManager or when KTM calls the ResourceManagerNotification callback routine.

While a resource manager is processing a notification, it might use the key as a pointer to a temporary memory allocation, and it might use the reference count to determine when it should deallocate the memory.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<= DISPATCH_LEVEL"

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

TmDereferenceEnlistmentKey

ZwCreateEnlistment

ZwOpenEnlistment

TmRenameTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRenameTransactionManager** routine changes the identity of the transaction manager object that is stored in the CLFS log file stream contained in the log file name.

> **Warning** Changing the identity of the transaction manger object might break any cross-log transactional links that may exist.

Syntax

```
NTSTATUS TmRenameTransactionManager(
  PUNICODE_STRING LogFileName,
  LPGUID          ExistingTransactionManagerGuid
);
```

Parameters

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of a CLFS log file stream to be associated with the transaction manager object.

`ExistingTransactionManagerGuid`

A pointer to a GUID structure that represents the current name of the transaction manager object.

Return Value

The **TmRenameTransactionManager** routine returns an NTSTATUS value. If the routine fails, it returns one of the following error codes:

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Universal
Header	wdm.h
IRQL	"<= APC_LEVEL"

TmRequestOutcomeEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRequestOutcomeEnlistment** routine asks KTM to try to provide an immediate outcome (commit or rollback) for the transaction that is associated with a specified enlistment.

Syntax

```
NTSTATUS TmRequestOutcomeEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmRequestOutcomeEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_ENLISTED	The specified enlistment is a superior enlistment.

The routine might return other NTSTATUS values.

Remarks

The **TmRequestOutcomeEnlistment** routine asks KTM to try to provide an immediate outcome (result) for the

transaction. A resource manager can call **TmRequestOutcomeEnlistment** after it has called TmPrepareComplete or ZwPrepareComplete, if it later discovers that it cannot wait for an outcome because, for example, a surprise-removal of the disk has occurred. KTM might be able to force a rollback if all resource managers have not finished their prepare operations.

For information about when to use KTM's **Tm*Xxx*** routines instead of Zw*Xxx* routines, see Using TmXxx Routines.

For more information about **TmCreateEnlistment**, see Creating a Resource Manager.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

TmPrepareComplete

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

TmRollbackComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRollbackComplete** routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.

Syntax

```
NTSTATUS TmRollbackComplete(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmRollbackComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_REQUESTED	The transaction or its enlistment is not in the correct state.

The routine might return other NTSTATUS values.

Remarks

The **TmRollbackComplete** routine is a pointer-based version of the ZwRollbackComplete routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwRollbackComplete

TmRollbackEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRollbackEnlistment** routine rolls back a specified enlistment.

Syntax

```
NTSTATUS TmRollbackEnlistment(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmRollbackEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment cannot be rolled back.

The routine might return other NTSTATUS values.

Remarks

The **TmRollbackEnlistment** routine is a pointer-based version of the ZwRollbackEnlistment routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwRollbackEnlistment

TmRollbackTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **TmRollbackTransaction** routine initiates a rollback operation for a specified transaction.

Syntax

```
NTSTATUS TmRollbackTransaction(
  PKTRANSACTION Transaction,
  BOOLEAN       Wait
);
```

Parameters

`Transaction`

A pointer to a transaction object. To obtain this pointer, your component must call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateTransaction or ZwOpenTransaction provided.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is set to **TRUE**, the call does not return until the rollback operation is complete.

Return Value

TmRollbackTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_ALREADY_COMMITTED	The transaction cannot be rolled back because it has already been committed.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The transaction has not been committed but its current state does not permit rollback.
STA TUS _PE NDI NG	Rollback notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

The **TmRollbackTransaction** routine is a pointer-based version of the ZwRollbackTransaction routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ZwCreateTransaction

ZwOpenTransaction

ZwRollbackTransaction

TmSinglePhaseReject function

1/25/2019 • 2 minutes to read • Edit Online

The **TmSinglePhaseReject** routine informs KTM that the calling resource manager will not support a single-phase commit operation for a specified enlistment.

Syntax

```
NTSTATUS TmSinglePhaseReject(
  PKENLISTMENT   Enlistment,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`Enlistment`

A pointer to an enlistment object. Your component can receive this pointer as input to a ResourceManagerNotification callback routine. Alternatively, your component can call ObReferenceObjectByHandle and supply the object handle that a previous call to ZwCreateEnlistment, TmCreateEnlistment, or ZwOpenEnlistment provided.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

TmSinglePhaseReject returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return the following value:

RETURN CODE	DESCRIPTION
STATUS_TRANSACTION_NOT_REQUESTED	The transaction or its enlistment is not in the correct state.

The routine might return other NTSTATUS values.

Remarks

The **TmSinglePhaseReject** routine is a pointer-based version of the ZwSinglePhaseReject routine.

For information about when to use KTM's **Tm*Xxx*** routines instead of **Zw*Xxx*** routines, see Using TmXxx Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ObReferenceObjectByHandle

ResourceManagerNotification

TmCreateEnlistment

ZwCreateEnlistment

ZwOpenEnlistment

ZwSinglePhaseReject

TRANSLATE_BUS_ADDRESS callback function

1/25/2019 • 2 minutes to read • Edit Online

The *TranslateBusAddress* routine translates addresses on the parent bus to logical addresses.

Syntax

```
TRANSLATE_BUS_ADDRESS TranslateBusAddress;

BOOLEAN TranslateBusAddress(
  PVOID Context,
  PHYSICAL_ADDRESS BusAddress,
  ULONG Length,
  PULONG AddressSpace,
  PPHYSICAL_ADDRESS TranslatedAddress
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the BUS_INTERFACE_STANDARD structure for the interface.

`BusAddress`

The bus-relative address to be translated.

`Length`

The length, in bytes, of the input memory that *BusAddress* points to.

`AddressSpace`

On input, the address space in which the device's hardware address resides. A value of 0x0 signifies memory space, and a value of 0x1 signifies port I/O space. On output, *AddressSpace* indicates the address space in which the *TranslatedAddress* resides.

`TranslatedAddress`

The translated (logical) address that corresponds to the bus-relative address that the caller provides in *BusAddress*.

Return Value

The *TranslateBusAddress* routine returns **TRUE** if the translation operation succeeded and **FALSE** otherwise.

Requirements

Target Platform	Desktop

Header	wdm.h (include Ntddk.h)
IRQL	PASSIVE_LEVEL

See Also

BUS_INTERFACE_STANDARD

VslCreateSecureSection function

1/25/2019 • 2 minutes to read • Edit Online

This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.

Syntax

```
NTSTATUS VslCreateSecureSection(
  PHANDLE   Handle,
  PEPROCESS TargetProcess,
  PMDL      Mdl,
  ULONG     DevicePageProtection,
  ULONG     Attributes
);
```

Parameters

`Handle`

`TargetProcess`

`Mdl`

`DevicePageProtection`

`Attributes`

Return Value

}

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h (include Wdm.h, Wdm.h)

VslDeleteSecureSection function

1/25/2019 • 2 minutes to read • Edit Online

This material is not yet available. This placeholder topic is provided as an example of documentation that may be included in a later release.

Syntax

```
NTSTATUS VslDeleteSecureSection(
  HANDLE GlobalHandle
);
```

Parameters

`GlobalHandle`

Return Value

None

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	wdm.h (include Wdm.h, Wdm.h)

WmiQueryTraceInformation function

1/25/2019 • 3 minutes to read • Edit Online

The **WmiQueryTraceInformation** routine returns information about a WMI event trace.

Syntax

```
NTSTATUS WmiQueryTraceInformation(
  TRACE_INFORMATION_CLASS TraceInformationClass,
  PVOID                   TraceInformation,
  ULONG                   TraceInformationLength,
  PULONG                  RequiredLength,
  PVOID                   Buffer
);
```

Parameters

`TraceInformationClass`

Specifies a TRACE_INFORMATION_CLASS enumerator that indicates the type of information to return about an event trace.

`TraceInformation`

A pointer to a caller-allocated output buffer where the routine returns the event trace information specified by *TraceInformationClass*.

`TraceInformationLength`

Specifies the size, in bytes, of the *TraceInformation* buffer.

`RequiredLength`

A pointer to the value returned by the routine that specifies the required size, in bytes, of the *TraceInformation* buffer. The caller should set *RequiredLength* to **NULL** if it does not use the required length information.

`Buffer`

A pointer to the query-specific input information that a caller supplies. If caller-supplied information is not required, the caller should set *Buffer* to **NULL**.

Return Value

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	Success

STATUS_INFO_LENGTH_MISMATCH	The size of the *TraceInformation* buffer is not equal to the required size for the specified event trace information.
STATUS_INVALID_HANDLE	The trace handle specified by the **HistoricalContext** member of the (PWNODE_HEADER)*Buffer* is not valid.
STATUS_INVALID_INFO_CLASS	The specified type of event trace information is not valid.
STATUS_INVALID_PARAMETER	The name of the event trace, supplied with a query to return a trace handle given its name, is not valid.
STATUS_INVALID_PARAMETER_MIX	The caller did not supply the information required for the specified event trace information.

STATUS_MORE_ENTRIES	The *TraceInformation* buffer is not large enough to hold an array of all the valid event trace handles.
STATUS_NOT_FOUND	A global logger was not found.

Remarks

For each type of event trace information specified by *TraceInformationClass*, the following table provides:

- Input requirements
- Information that **WmiQueryTraceInformation** returns in the *TraceInformation* buffer

VALUE OF*TRACECLASSINFORMATION*	INPUT REQUIREMENTS	INFORMATION RETURNED
TraceIdClass	*TraceInformationLength* is equal to the value of **sizeof**(ULONG). The size, in bytes of the *TraceInformation* buffer is greater than or equal to the value of **sizeof**(ULONG). The **HistoricalContext** member of (PWNODE_HEADER)*Buffer*specifies an event trace handle.	*(PULONG)*TraceInformation* is set to the logger ID of the event trace handle.
TraceHandleClass	*TraceInformationLength* is equal to the value of **sizeof**(TRACEHANDLE). The size, in bytes of the *TraceInformation* buffer must be greater than or equal to the value of **sizeof**(TRACEHANDLE). *(PULONG)*Buffer* is set to a logger ID.	*(PTRACEHANDLE)*TraceInformation* is set to an event trace handle for the specified logger. If the specified logger ID is zero, an event trace handle for the kernel logger is returned.

TraceEnableFlagsClass	*TraceInformationLength* is greater than or equal to the value of **sizeof**(ULONG). The size, in bytes of the *TraceInformation* buffer must be greater than or equal to the value of **sizeof**(ULONG). The **HistoricalContext** member of (PWNODE_HEADER)*Buffer*specifies an event trace handle.	*(PULONG)*TraceInformation* is set to the enable flags that are set for the specified event trace handle.
TraceEnableLevelClass	*TraceInformationLength* is set greater than or equal to the value of **sizeof**(ULONG). The size, in bytes of the *TraceInformation* buffer must be greater than or equal to the value of **sizeof**(ULONG). The **HistoricalContext** member of (PWNODE_HEADER)*Buffer*specifies an event trace handle.	*(PULONG)*TraceInformation* is set to the level for the specified event trace handle.
GlobalLoggerHandleClass	*TraceInformationLength* is equal to the value of **sizeof**(TRACEHANDLE). The size, in bytes of the *TraceInformation* buffer must be greater than or equal to the value of **sizeof**(TRACEHANDLE).	*(PTRACEHANDLE)*TraceInformation* is set to an event trace handle for the global logger.
EventLoggerHandleClass	For internal use only.	For internal use only.
AllLoggerHandlesClass	*TraceInformationLength* is set to the size, bytes, of an array of *m* TRACEHANDLE values. The size, in bytes of the *TraceInformation* buffer must be greater than or equal to the value of (*m****sizeof**(TRACEHANDLE)).	The *TraceInformation* buffer contains an array of *n* trace handles, where *n* is the minimum of *m*, the number of caller-supplied event trace handles, and the number of valid event trace handles. The routine returns a status of STATUS_MORE_ENTRIES if the *TraceInformation* buffer is too small to hold all trace handles.
TraceHandleByNameClass	*TraceInformationLength* is set to the value of **sizeof**(TRACEHANDLE). The size, in bytes, of the *TraceInformation* buffer must be greater than or equal to the value of **sizeof**(TRACEHANDLE). (PUNICODE_STRING)*Buffer* specifies a friendly trace name in Unicode format.	*(PTRACEHANDLE)*TraceInformation* is set to the event trace handle associated with the specified friendly name.

If the caller supplies a non-**NULL** *RequiredLength* pointer, **WmiQueryTraceInformation** also returns the required length for the specified event trace information.

WmiQueryTraceInformation runs at the IRQL of the caller.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

IoWmiWriteEvent

TRACE_INFORMATION_CLASS

WmiFireEvent

WmiTraceMessage

WmiTraceMessageVa

WmiTraceMessage function

1/25/2019 • 2 minutes to read • Edit Online

The **WmiTraceMessage** routine adds a message to the output log of a WPP software tracing session.

Syntax

```
NTSTATUS WmiTraceMessage(
  TRACEHANDLE LoggerHandle,
  ULONG       MessageFlags,
  LPCGUID     MessageGuid,
  USHORT      MessageNumber,
  ...
);
```

Parameters

`LoggerHandle`

Specifies a trace handle for a software tracing session.

`MessageFlags`

Specifies a bitwise OR of one or more message flags. See the Remarks section for details.

`MessageGuid`

Specifies a GUID that identifies the class of software trace messages.

`MessageNumber`

Identifies the message subtype. The meaning of subtypes is specific to the software trace class and the provider.

`arg5`

Return Value

WmiTraceMessage returns one of the following values:

RETURN CODE	DESCRIPTION
STA **TUS** **_SU** **CCE** **SS**	The operation completed successfully.

STATUS_INVALID_HANDLE	*LoggerHandle* is not a valid software trace handle.
STATUS_NO_MEMORY	There is insufficient buffer memory to log the message. See the Remarks section.
Other NTSTATUS value	An internal error occurred.

Remarks

A caller can use **WmiTraceMessage** or WmiTraceMessageVa to add a message to the output log of a WPP software tracing session. **WmiTraceMessage** simplifies a caller's code by handling the variable list mechanism before calling **WmiTraceMessageVa**.

A caller can set the following message flags:

FLAG	DESCRIPTION
TRACE_MESSAGE_SEQUENCE	Include a sequence number in the message. Message sequencing can only be used if it is set for the software tracing session specified by *LoggerHandle*.
TRACE_MESSAGE_GUID	*MessageGuid* specifies a GUID that identifies a software trace class. This flag must be set.
TRACE_MESSAGE_TIMESTAMP	Include a time stamp in the message.
TRACE_MESSAGE_PERFORMANCE_TIMESTAMP	This flag is not implemented and is obsolete. It must not be used.
TRACE_MESSAGE_SYSTEMINFO	Include the thread identifier (TID) and process identifier (PID) in the message.

A message will not be logged if one of the following is true:

- The total size, in bytes, of the message data and the message header is greater than the size of individual message buffers allocated for a software tracing session. (The maximum message header size is 48 bytes.)

- All message buffers allocated to software tracing session are full.

WmiTraceMessage runs at the IRQL of the caller.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

IoWmiWriteEvent

TRACE_INFORMATION_CLASS

WmiFireEvent

WmiQueryTraceInformation

WmiTraceMessageVa

WmiTraceMessageVa function

1/25/2019 • 2 minutes to read • Edit Online

The **WmiTraceMessageVa** routine adds a message to the output log of a WPP software tracing session.

Syntax

```
NTSTATUS WmiTraceMessageVa(
  TRACEHANDLE LoggerHandle,
  ULONG       MessageFlags,
  LPCGUID     MessageGuid,
  USHORT      MessageNumber,
  va_list     MessageArgList
);
```

Parameters

`LoggerHandle`

Specifies a trace handle for a software tracing session.

`MessageFlags`

Specifies a bitwise OR of one or more message flags. See the Remarks section for details.

`MessageGuid`

Specifies a GUID that identifies a software trace class.

`MessageNumber`

Identifies the message subtype. The meaning of subtypes is specific to the software trace class and the provider.

`MessageArgList`

Provides a required list of message parameters that specify a set of message parts. The parameters are organized as a sequence of parameter pairs, where each pair specifies one part of the complete message. Each parameter pair consists of a PVOID pointer to data, followed immediately by a ULONG value that specifies the length of the data. The parameter list must be terminated by a **NULL** PVOID pointer followed by a ULONG(0).

Return Value

WmiTraceMessageVa returns one of the following values:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The operation completed successfully.

STATUS_INVALID_HANDLE	*LoggerHandle* is not a valid software trace handle.
STATUS_NO_MEMORY	There is insufficient buffer memory to log the message. See the Remarks section for more information.
OtherNTSTATUSvalue	An internal error occurred.

Remarks

A caller can use WmiTraceMessage or **WmiTraceMessageVa** to add a message to the output log of a WPP software tracing session. **WmiTraceMessage** simplifies a caller's code by handling the variable list mechanism before calling **WmiTraceMessageVa**.

A caller can set the following message flags:

FLAG	DESCRIPTION
TRACE_MESSAGE_SEQUENCE	Include a sequence number in the message. Message sequencing can only be used if it is enabled for the software tracing session specified by *LoggerHandle*.
TRACE_MESSAGE_GUID	*MessageGuid* specifies a GUID that identifies a software trace class. This flag must be set.
TRACE_MESSAGE_TIMESTAMP	Include a time stamp in the message.
TRACE_MESSAGE_PERFORMANCE_TIMESTAMP	This flag is not implemented and is obsolete. It must not be used.
TRACE_MESSAGE_SYSTEMINFO	Include the thread identifier (TID) and process identifier (PID) in the message.

WmiTraceMessageVa does not log the message if one of the following is true:

- The total size, in bytes, of the message data and the message header is greater than the size of individual message buffers allocated for a software tracing session. (The maximum message header size is 48 bytes.)

- All message buffers allocated to software tracing session are full.

WmiTraceMessageVa runs at the IRQL of the caller.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

IoWmiWriteEvent

TRACE_INFORMATION_CLASS

WmiFireEvent

WmiQueryTraceInformation

WmiTraceMessage

WRITE_PORT_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_UCHAR** routine writes a number of bytes from a buffer to the specified port.

Syntax

```
NTHALAPI VOID WRITE_PORT_BUFFER_UCHAR(
  PUCHAR Port,
  PUCHAR Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of UCHAR values is to be written.

`Count`

Specifies the number of bytes to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **WRITE_PORT_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_ULONG** routine writes a number of ULONG values from a buffer to the specified port address.

Syntax

```
NTHALAPI VOID WRITE_PORT_BUFFER_ULONG(
  PULONG Port,
  PULONG Buffer,
  ULONG  Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of ULONG values is to be written.

`Count`

Specifies the number of ULONG values to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **WRITE_PORT_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

WRITE_PORT_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_BUFFER_USHORT** routine writes a number of USHORT values from a buffer to the specified port address.

Syntax

```
NTHALAPI VOID WRITE_PORT_BUFFER_USHORT(
  PUSHORT Port,
  PUSHORT Buffer,
  ULONG   Count
);
```

Parameters

`Port`

Pointer to the port, which must be a mapped memory range in I/O space.

`Buffer`

Pointer to a buffer from which an array of USHORT values is to be written.

`Count`

Specifies the number of USHORT values to be written to the port.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **WRITE_PORT_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h)
Library	Hal.lib

IRQL	Any level (see Remarks section)

WRITE_PORT_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_UCHAR** routine writes a byte to the specified port address.

Syntax

```
NTHALAPI VOID WRITE_PORT_UCHAR(
  PUCHAR Port,
  UCHAR  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_UCHAR** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_ULONG** routine writes a ULONG value to the specified port address.

Syntax

```
NTHALAPI VOID WRITE_PORT_ULONG(
  PULONG Port,
  ULONG  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_ULONG** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_PORT_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_PORT_USHORT** routine writes a USHORT value to the specified port address.

Syntax

```
NTHALAPI VOID WRITE_PORT_USHORT(
  PUSHORT Port,
  USHORT  Value
);
```

Parameters

`Port`

`Value`

Return Value

None

Remarks

Callers of **WRITE_PORT_USHORT** can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h)
Library	Hal.lib
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_UCHAR** routine writes a number of bytes from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of UCHAR values is to be written.

`Count`

Specifies the number of bytes to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **WRITE_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_ULONG** routine writes a number of ULONG values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of ULONG values is to be written.

`Count`

Specifies the number of ULONG values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **WRITE_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_ULONG64 function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_ULONG64** function writes a number of ULONG64 values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_ULONG64(
  volatile ULONG64 *Register,
  PULONG64         Buffer,
  ULONG            Count
);
```

Parameters

`Register`

A pointer to the register, which must be a mapped range in memory space.

`Buffer`

A pointer to a buffer into which an array of ULONG64 values is to be written.

`Count`

Specifies the number of ULONG64 values to write to the register.

Return Value

This function does not return a value.

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

For more information, see Reading and Writing to Device Registers in UMDF 1.x Drivers.

Requirements

Minimum supported client	64-bit Windows
Target Platform	Desktop
Header	wdm.h (include Wdm.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.exe

WRITE_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_USHORT** routine writes a number of USHORT values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of USHORT values is to be written.

`Count`

Specifies the number of USHORT values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **WRITE_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_UCHAR** routine writes a byte to the specified address.

Syntax

```
void WRITE_REGISTER_UCHAR(
  volatile UCHAR *Register,
  UCHAR          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_ULONG** routine writes a ULONG value to the specified address.

Syntax

```
void WRITE_REGISTER_ULONG(
  volatile ULONG *Register,
  ULONG          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_USHORT** routine writes a USHORT value to the specified address.

Syntax

```
void WRITE_REGISTER_USHORT(
  volatile USHORT *Register,
  USHORT          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

ZwClose function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwClose** routine closes an object handle.

Syntax

```
NTSYSAPI NTSTATUS ZwClose(
  HANDLE Handle
);
```

Parameters

`Handle`

Handle to an object of any type.

Return Value

ZwClose returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. In particular, it returns STATUS_INVALID_HANDLE if *Handle* is not a valid handle, or STATUS_HANDLE_NOT_CLOSABLE if the calling thread does not have permission to close the handle.

Remarks

ZwClose is a generic routine that operates on any type of object.

Closing an open object handle causes that handle to become invalid. The system also decrements the handle count for the object and checks whether the object can be deleted. The system does not actually delete the object until all of the object's handles are closed and no referenced pointers remain.

A driver must close every handle that it opens as soon as the handle is no longer required. Kernel handles, which are those that are opened by a system thread or by specifying the OBJ_KERNEL_HANDLE flag, can be closed only when the previous processor mode is **KernelMode**. This requirement applies both to system threads and to dispatch routines for IRPs that were issued from kernel mode. (For more information about the previous processor mode, see ExGetPreviousMode.) For example, a handle that ZwCreateKey returns to a DriverEntry routine cannot subsequently be closed by the same driver's dispatch routines. A **DriverEntry** routine runs in a system process, whereas dispatch routines usually run either in the context of the thread issuing the current I/O request, or, for lower-level drivers, in an arbitrary thread context.

A nonkernel handle can be closed only if one of two conditions is met: The previous processor mode is **KernelMode**, or the calling thread has sufficient permission to close the handle. An example of the latter occurs when the calling thread is the one that created the handle.

Callers of **ZwClose** should not assume that this routine automatically waits for all I/O to complete prior to returning.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtClose**" instead of "**ZwClose**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can

behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateDirectoryObject

ZwCreateFile

ZwCreateKey

ZwOpenKey

ZwOpenSection

ZwCommitComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitComplete** routine notifies KTM that the calling resource manager has finished committing a transaction's data.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCommitComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwCommitComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.
STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwCommitComplete** after it has finished servicing a TRANSACTION_NOTIFY_COMMIT notification.

For more information about **ZwCommitComplete**, see Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwCommitEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitEnlistment** routine initiates the commit operation for a specified enlistment's transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCommitEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwCommitEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.
STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_COMMIT_COMPLETE notifications.
STATUS_TRANSACTION_REQUEST_NOT_VALID	The enlistment's transaction is not in a state that allows it to be committed.

STA TUS _TR AN SAC TIO N_N OT_ ACT IVE	The commit operation for this transaction has already been started.
STA TUS _TR AN SAC TIO N_A LRE AD Y_A BO RTE D	The transaction cannot be committed because it has been rolled back.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwCommitEnlistment**.

Callers of **ZwCommitEnlistment** must register to receive TRANSACTION_NOTIFY_COMMIT_COMPLETE notifications.

The **ZwCommitEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_COMMIT notifications to all resource managers that have enlisted in the transaction.

For more information about **ZwCommitEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwCommitTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCommitTransaction** routine initiates a commit operation for a specified transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCommitTransaction(
  HANDLE  TransactionHandle,
  BOOLEAN Wait
);
```

Parameters

`TransactionHandle`

A handle to a transaction object. Your component receives this handle from ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_COMMIT access to the object.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is **TRUE**, the call returns after the commit operation is complete.

Return Value

ZwCommitTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that was specified for the *TransactionHandle* parameter is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The specified transaction object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the transaction object.
STA TUS _TR AN SAC TIO N_S UPE RIO R_E XIS TS	The caller cannot commit the transaction because a superior transaction manager exists.
STA TUS _TR AN SAC TIO N_A LRE AD Y_A BO RTE D	The transaction cannot be committed because it has been rolled back.
STA TUS _TR AN SAC TIO N_A LRE AD Y_C OM MIT TED	The transaction is already committed.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The commit operation for this transaction has already been started.
STA TUS _PE NDI NG	Commit notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

For more information about how transaction clients should use the **ZwCommitTransaction** routine, see Creating a Transactional Client.

For more information about commit operations, see Handling Commit Operations.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmCommitTransaction

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

ZwCreateDirectoryObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwCreateDirectoryObject** routine creates or opens an object-directory object.

Syntax

```
NTSYSAPI NTSTATUS ZwCreateDirectoryObject(
  PHANDLE            DirectoryHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`DirectoryHandle`

Pointer to a HANDLE variable that receives a handle to the object directory.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects (see ACCESS_MASK), the caller can specify one or more of the following access rights, which are specific to object directories:

ACCESS_MASK FLAG	TYPE OF ACCESS
DIRECTORY_QUERY	Query
DIRECTORY_TRAVERSE	Name lookup
DIRECTORY_CREATE_OBJECT	Name creation
DIRECTORY_CREATE_SUBDIRECTORY	Subdirectory creation
DIRECTORY_ALL_ACCESS	All of the preceding types

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that contains the object's attributes, which you must have already initialized by calling InitializeObjectAttributes.

Return Value

ZwCreateDirectoryObject returns an NTSTATUS value. Possible return values include:

Remarks

Once the handle pointed to by *DirectoryHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles.

Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

The system uses object directories to organize other types of objects, such as device objects. For more information, see Object Directories.

Note that the system does not use object directory objects to represent file-system directories, which are represented instead as file objects.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateDirectoryObject**" instead of "**ZwCreateDirectoryObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateEnlistment function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateEnlistment** routine creates a new enlistment object for a transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCreateEnlistment(
  PHANDLE            EnlistmentHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             ResourceManagerHandle,
  HANDLE             TransactionHandle,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              CreateOptions,
  NOTIFICATION_MASK  NotificationMask,
  PVOID              EnlistmentKey
);
```

Parameters

`EnlistmentHandle`

A pointer to a caller-allocated variable that receives a handle to the new enlistment object if the call to **ZwCreateEnlistment** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the enlistment object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following access right flags for enlistment objects:

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
ENLISTMENT_QUERY_INFORMATION	Query information about the enlistment (see ZwQueryInformationEnlistment).
ENLISTMENT_SET_INFORMATION	Set information for the enlistment (see ZwSetInformationEnlistment).
ENLISTMENT_RECOVER	Recover the enlistment (see ZwRecoverEnlistment).
ENLISTMENT_SUBORDINATE_RIGHTS	Perform operations that a resource manager that is not superior performs (see ZwRollbackEnlistment, ZwPrePrepareComplete, ZwPrepareComplete, ZwCommitComplete, ZwRollbackComplete, ZwSinglePhaseReject, ZwReadOnlyEnlistment).
ENLISTMENT_SUPERIOR_RIGHTS	Perform operations that a superior transaction manager must perform (see ZwPrepareEnlistment, ZwPrePrepareEnlistment, ZwCommitEnlistment).

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the ACCESS_MASK

reference page. You can also combine these bitmaps together with additional flags from the previous table. The following table shows how the bitmaps correspond to specific access rights.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
ENLISTMENT_GENERIC_READ	STANDARD_RIGHTS_READ and ENLISTMENT_QUERY_INFORMATION
ENLISTMENT_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, ENLISTMENT_SET_INFORMATION, ENLISTMENT_RECOVER, ENLISTMENT_REFERENCE, ENLISTMENT_SUBORDINATE_RIGHTS, and ENLISTMENT_SUPERIOR_RIGHTS
ENLISTMENT_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, ENLISTMENT_RECOVER, ENLISTMENT_SUBORDINATE_RIGHTS, and ENLISTMENT_SUPERIOR_RIGHTS
ENLISTMENT_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, ENLISTMENT_GENERIC_READ, ENLISTMENT_GENERIC_WRITE, and ENLISTMENT_GENERIC_EXECUTE

`ResourceManagerHandle`

A handle to the caller's resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager.

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. KTM adds this transaction to the list of transactions that the calling resource manager is handling.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`CreateOptions`

Enlistment option flags. The following table contains the only available flag.

CREATEOPTIONS FLAG	MEANING
ENLISTMENT_SUPERIOR	The caller is enlisting as a superior transaction manager for the specified transaction.

This parameter is optional and can be zero.

`NotificationMask`

A bitwise OR of TRANSACTION_NOTIFY_*XXX* values that are defined in Ktmtypes.h. This mask specifies the types of transaction notifications that KTM sends to the caller.

`EnlistmentKey`

A pointer to caller-defined information that uniquely identifies the enlistment. The resource manager receives this pointer when it calls ZwGetNotificationResourceManager or when KTM calls the ResourceManagerNotification

callback routine. The resource manager can maintain a reference count for this key by calling TmReferenceEnlistmentKey and TmDereferenceEnlistmentKey. This parameter is optional and can be **NULL**.

Return Value

ZwCreateEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_INVALID_PARAMETER	The *CreateOptions* or *NotificationMask* parameter's value is invalid, or KTM could not find the transaction that the *TransactionHandle* parameter specifies.
STATUS_INSUFFICIENT_RESOURCES	A memory allocation failed.
STATUS_TRANSACTION_MANAGER_NOT_ONLINE	The enlistment failed because KTM or the resource manager is not in an operational state.

STATUS_TRANSACTION_NOT_ACTIVE	The enlistment failed because the transaction that the *TransactionHandle* parameter specifies is not active.
STATUS_TRANSACTION_SUPERIOR_EXISTS	The caller tried to register as a superior transaction manager but a superior transaction manager already exists.
STATUS_TM_VOLATILE	The caller is trying to register as a superior transaction manager, but the caller's resource manager object is volatile while the associated transaction manager object is not volatile.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

A resource manager calls **ZwCreateEnlistment** to enlist in a transaction.

Resource managers that are not superior must include the ENLISTMENT_SUBORDINATE_RIGHTS flag in their access mask.

Superior transaction managers must include the ENLISTMENT_SUPERIOR_RIGHTS flag in their access masks. Typically, a superior transaction manager includes code that calls ZwRollbackEnlistment, so it must also include the ENLISTMENT_SUBORDINATE_RIGHTS flag.

A resource manager that calls **ZwCreateEnlistment** must eventually call ZwClose to close the object handle.

Your resource manager can use the *EnlistmentKey* parameter to assign a unique value to each enlistment, such as a

pointer to a data structure that contains information about the enlistment. For example, if the resource manager stores the enlistment object's handle in the structure, the resource manager can do the following:

1. Call ZwGetNotificationResourceManager to obtain a notification.
2. Obtain the enlistment key value from the TRANSACTION_NOTIFICATION structure.
3. Use the enlistment key to find the stored enlistment object handle.
4. Call routines that require the enlistment handle as input, such as ZwCommitComplete or ZwRollbackComplete.

For more information about **ZwCreateEnlistment**, see Creating a Resource Manager and Creating a Superior Transaction Manager.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

TRANSACTION_NOTIFICATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCommitComplete

ZwCommitEnlistment

ZwCreateResourceManager

ZwCreateTransaction

ZwGetNotificationResourceManager

ZwOpenEnlistment

ZwOpenResourceManager

ZwOpenTransaction

ZwPrePrepareComplete

ZwPrePrepareEnlistment

ZwPrepareComplete

ZwPrepareEnlistment

ZwQueryInformationEnlistment

ZwReadOnlyEnlistment

ZwRecoverEnlistment

ZwRollbackComplete

ZwRollbackEnlistment

ZwSetInformationEnlistment

ZwSinglePhaseReject

ZwCreateFile function

1/25/2019 • 16 minutes to read • Edit Online

The **ZwCreateFile** routine creates a new file or opens an existing file.

Syntax

```
NTSYSAPI NTSTATUS ZwCreateFile(
  PHANDLE            FileHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PIO_STATUS_BLOCK   IoStatusBlock,
  PLARGE_INTEGER     AllocationSize,
  ULONG              FileAttributes,
  ULONG              ShareAccess,
  ULONG              CreateDisposition,
  ULONG              CreateOptions,
  PVOID              EaBuffer,
  ULONG              EaLength
);
```

Parameters

`FileHandle`

A pointer to a HANDLE variable that receives a handle to the file.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects, the caller can specify any of the following access rights, which are specific to files.

ACCESS_MASK FLAG	ALLOWS CALLER TO DO THIS
FILE_READ_DATA	Read data from the file.
FILE_READ_ATTRIBUTES	Read the attributes of the file. (For more information, see the description of the *FileAttributes* parameter.)
FILE_READ_EA	Read the extended attributes (EAs) of the file. This flag is irrelevant for device and intermediate drivers.
FILE_WRITE_DATA	Write data to the file.
FILE_WRITE_ATTRIBUTES	Write the attributes of the file. (For more information, see the description of the *FileAttributes* parameter.)
FILE_WRITE_EA	Change the extended attributes (EAs) of the file. This flag is irrelevant for device and intermediate drivers.
FILE_APPEND_DATA	Append data to the file.

FILE_EXECUTE	Use system paging I/O to read data from the file into memory. This flag is irrelevant for device and intermediate drivers.

> **Note** Do not specify FILE_READ_DATA, FILE_WRITE_DATA, FILE_APPEND_DATA, or FILE_EXECUTE when you create or open a directory.

The caller can only specify a generic access right, GENERIC_*XXX*, for a file, not a directory. Generic access rights correspond to specific access rights as shown in the following table.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
GENERIC_READ	STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES, FILE_READ_EA, and SYNCHRONIZE.
GENERIC_WRITE	STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA, FILE_APPEND_DATA, and SYNCHRONIZE.
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, FILE_EXECUTE, FILE_READ_ATTRIBUTES, and SYNCHRONIZE. This value is irrelevant for device and intermediate drivers.
GENERIC_ALL	FILE_ALL_ACCESS.

For example, if you specify GENERIC_READ for a file object, the routine maps this value to the FILE_GENERIC_READ bitmask of specific access rights. In the preceding table, the specific access rights that are listed for GENERIC_READ correspond to the access flags that are contained in the FILE_GENERIC_READ bitmask. If the file is actually a directory, the caller can also specify the following generic access rights.

DESIREDACCESS FLAG	ALLOWS CALLER TO DO THIS
FILE_LIST_DIRECTORY	List the files in the directory.
FILE_TRAVERSE	Traverse the directory, in other words, include the directory in the path of a file.

For more information about access rights, see ACCESS_MASK. `ObjectAttributes` A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. `IoStatusBlock` A pointer to an IO_STATUS_BLOCK structure that receives the final completion status and other information about the requested operation. In particular, the **Information** member receives one of the following values:

- FILE_CREATED
- FILE_OPENED
- FILE_OVERWRITTEN
- FILE_SUPERSEDED
- FILE_EXISTS
- FILE_DOES_NOT_EXIST

`AllocationSize` A pointer to a LARGE_INTEGER that contains the initial allocation size, in bytes, for a file that is created or overwritten. If *AllocationSize* is **NULL**, no allocation size is specified. If no file is created or overwritten, *AllocationSize* is ignored. `FileAttributes` Specifies one or more FILE_ATTRIBUTE_*XXX* flags, which represent the file attributes to set if you create or overwrite a file. The caller usually specifies FILE_ATTRIBUTE_NORMAL, which sets

the default attributes. For a list of valid FILE_ATTRIBUTE_*XXX* flags, see the CreateFile routine in the Microsoft Windows SDK documentation. If no file is created or overwritten, *FileAttributes* is ignored. `ShareAccess` Type of share access, which is specified as zero or any combination of the following flags.

***ShareAccess* flag**	**Allows other threads to do this**
FILE_SHARE_READ	Read the file
FILE_SHARE_WRITE	Write the file
FILE_SHARE_DELETE	Delete the file

Device and intermediate drivers usually set *ShareAccess* to zero, which gives the caller exclusive access to the open file. `CreateDisposition` Specifies the action to perform if the file does or does not exist. *CreateDisposition* can be one of the values in the following table.

***CreateDisposition* value**	**Action if file exists**	**Action if file does not exist**
FILE_SUPERSEDE	Replace the file.	Create the file.
FILE_CREATE	Return an error.	Create the file.
FILE_OPEN	Open the file.	Return an error.
FILE_OPEN_IF	Open the file.	Create the file.
FILE_OVERWRITE	Open the file, and overwrite it.	Return an error.
FILE_OVERWRITE_IF	Open the file, and overwrite it.	Create the file.

`CreateOptions` Specifies the options to apply when the driver creates or opens the file. Use one or more of the flags in the following table.

***CreateOptions* flag**	**Meaning**
FILE_DIRECTORY_FILE	The file is a directory. Compatible *CreateOptions* flags are FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT, and FILE_OPEN_BY_FILE_ID. The *CreateDisposition* parameter must be set to FILE_CREATE, FILE_OPEN, or FILE_OPEN_IF.
FILE_NON_DIRECTORY_FILE	The file is not a directory. The file object to open can represent a data file; a logical, virtual, or physical device; or a volume.
FILE_WRITE_THROUGH	System services, file-system drivers, and drivers that write data to the file must actually transfer the data to the file before any requested write operation is considered complete.
FILE_SEQUENTIAL_ONLY	All access to the file will be sequential.
FILE_RANDOM_ACCESS	Access to the file can be random, so no sequential read-ahead operations should be performed by file-system drivers or by the system.

FILE_NO_INTERMEDIATE_BUFFERING	The file cannot be cached or buffered in a driver's internal buffers. This flag is incompatible with the *DesiredAccess* parameter's FILE_APPEND_DATA flag.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file-position pointer. If this flag is set, the SYNCHRONIZE flag must be set in the *DesiredAccess* parameter.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the file are performed synchronously. Waits in the system that synchronize I/O queuing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file-position context. If this flag is set, the SYNCHRONIZE flag must be set in the *DesiredAccess* parameter.
FILE_CREATE_TREE_CONNECTION	Create a tree connection for this file in order to open it over the network. This flag is not used by device and intermediate drivers.
FILE_COMPLETE_IF_OPLOCKED	Complete this operation immediately with an alternate success code of STATUS_OPLOCK_BREAK_IN_PROGRESS if the target file is opportunistic locked (oplock), rather than blocking the caller's thread. If the file is oplocked, another caller already has access to the file. This flag is not used by device and intermediate drivers. For information about oplock, see Opportunistic Locks.
FILE_NO_EA_KNOWLEDGE	If the extended attributes (EAs) for an existing file being opened indicate that the caller must understand EAs to properly interpret the file, **ZwCreateFile** should return an error. This flag is irrelevant for device and intermediate drivers.
FILE_OPEN_REPARSE_POINT	Open a file with a reparse point and bypass normal reparse point processing for the file. For more information, see the following Remarks section. For information about reparse point, see Reparse Points.
FILE_DELETE_ON_CLOSE	The system deletes the file when the last handle to the file is passed to ZwClose. If this flag is set, the DELETE flag must be set in the *DesiredAccess* parameter.

FILE_OPEN_BY_FILE_ID	The file name that is specified by the *ObjectAttributes* parameter includes a binary 8-byte or 16-byte file reference number or object ID for the file, depending on the file system as shown below. Optionally, a device name followed by a backslash character may proceed these binary values. For example, a device name will have the following format.
	
	`??\C:<8 bytes of binary FileID>`
	`\device\HardDiskVolume1<16 bytes of binary ObjectID>`
	
	• On NTFS, this can be a 8-byte or 16-byte reference number or object ID. A 16-byte reference number is the same as an 8-byte number padded with zeros.
	• On ReFS, this can be an 8-byte or 16-byte reference number. A 16-byte number is not related to an 8-byte number. Object IDs are not supported.
	• The FAT, ExFAT, UDFS, and CDFS file systems do not support this flag.
	
	This number is assigned by and specific to the particular file system.
	
	Note Because the filename field will partly contain a binary blob, it is incorrect to assume that this is a valid Unicode string, and more importantly may not be a null terminated string.

FILE_OPEN_FOR_BACKUP_INTENT	The file is being opened for backup intent. Therefore, the system should check for certain access rights and grant the caller the appropriate access to the file—before checking the *DesiredAccess* parameter against the file's security descriptor. This flag not used by device and intermediate drivers.
FILE_RESERVE_OPFILTER	This flag allows an application to request a Filter opportunistic lock (oplock) to prevent other applications from getting share violations. If there are already open handles, the create request will fail with STATUS_OPLOCK_NOT_GRANTED. For more information, see the following Remarks section. For information about oplock, see Opportunistic Locks.
FILE_OPEN_REQUIRING_OPLOCK	The file is being opened and an opportunistic lock (oplock) on the file is being requested as a single atomic operation. The file system checks for oplocks before it performs the create operation, and will fail the create with a return code of STATUS_CANNOT_BREAK_OPLOCK if the result would be to break an existing oplock. **Note** The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2 and later Windows operating systems.
FILE_SESSION_AWARE	The client opening the file or device is session aware and per session access is validated if necessary. **Note** The FILE_SESSION_AWARE flag is available starting withWindows 8.

`EaBuffer` For device and intermediate drivers, this parameter must be a **NULL** pointer. `EaLength` For device and intermediate drivers, this parameter must be zero. ## Return Value **ZwCreateFile** returns STATUS_SUCCESS on success or an appropriate NTSTATUS error code on failure. In the latter case, the caller can determine the cause of the failure by checking the *IoStatusBlock* parameter.

> **Note ZwCreateFile** might return STATUS_FILE_LOCK_CONFLICT as the return value or in the **Status** member of the **IO_STATUS_BLOCK** structure that is pointed to by the *IoStatusBlock* parameter. This would occur only if the NTFS log file is full, and an error occurs while **ZwCreateFile** tries to handle this situation.

Remarks

ZwCreateFile supplies a handle that the caller can use to manipulate a file's data, or the file object's state and attributes. For more information, see Using Files in a Driver.

Once the handle pointed to by *FileHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

There are two alternate ways to specify the name of the file to be created or opened with **ZwCreateFile**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *ObjectAttributes*.
2. As pathname relative to the directory file represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*.

Setting certain flags in the *DesiredAccess* parameter results in the following effects:

- For a caller to synchronize an I/O completion by waiting for the returned *FileHandle*, the SYNCHRONIZE flag must be set. Otherwise, a caller that is a device or intermediate driver must synchronize an I/O completion by using an event object.
- If the caller sets only the FILE_APPEND_DATA and SYNCHRONIZE flags, it can write only to the end of the file, and any offset information about write operations to the file is ignored. The file will automatically be extended as necessary for this type of operation.
- Setting the FILE_WRITE_DATA flag for a file also allows the caller to write beyond the end of the file. Again, the file is automatically extended as necessary.
- If the caller sets only the FILE_EXECUTE and SYNCHRONIZE flags, it cannot directly read or write any data to the file using the returned *FileHandle*. That is, all operations on the file occur through the system pager in response to instruction and data-access operations. Device and intermediate drivers should not set the FILE_EXECUTE flag.

The *ShareAccess* parameter determines whether separate threads can access the same file, possibly simultaneously. Provided that both callers have the appropriate access privileges, the file can be successfully opened and shared. If the original caller of **ZwCreateFile** does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, no other caller can open the file—that is, the original caller is granted exclusive access.

To successfully open a shared file, the *DesiredAccess* flags must be compatible with the *DesiredAccess* and *ShareAccess* flags of all the previous open operations that have not yet been released through . That is, the *DesiredAccess* specified to **ZwCreateFile** for a given file must not conflict with the accesses that other openers of the file have disallowed.

The *CreateDisposition* value FILE_SUPERSEDE requires that the caller have DELETE access to a existing file object. If so, a successful call to **ZwCreateFile** with FILE_SUPERSEDE on an existing file effectively deletes that file, and

then recreates it. This implies that, if the file has already been opened by another thread, it opened the file by specifying a *ShareAccess* parameter with the FILE_SHARE_DELETE flag set. Note that this type of disposition is consistent with the POSIX style of overwriting files.

The *CreateDisposition* values FILE_OVERWRITE_IF and FILE_SUPERSEDE are similar. If **ZwCreateFile** is called with a existing file and either of these *CreateDisposition* values, the file will be replaced.

Overwriting a file is semantically equivalent to a supersede operation, except for the following:

- The caller must have write access to the file, rather than delete access. This implies that, if the file has already been opened by another thread, it opened the file with the FILE_SHARE_WRITE flag set in the input *ShareAccess*.
- The specified file attributes are logically ORed with those already on the file. This implies that, if the file has already been opened by another thread, a subsequent caller of **ZwCreateFile** cannot disable existing *FileAttributes* flags but can enable additional flags for the same file. Note that this style of overwriting files is consistent with MS-DOS, Microsoft Windows 3.1, and OS/2.

The FILE_DIRECTORY_FILE *CreateOptions* value specifies that the file to be created or opened is a directory. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the given file to be opened is not a directory file, or if the caller specified an inconsistent *CreateOptions* or *CreateDisposition* value, the call to **ZwCreateFile** will fail.

The FILE_NO_INTERMEDIATE_BUFFERING *CreateOptions* flag prevents the file system from performing any intermediate buffering on behalf of the caller. Specifying this flag places the following restrictions on the caller's parameters to other **Zw*Xxx*File** routines.

- Any optional *ByteOffset* passed to ZwReadFile or ZwWriteFile must be a multiple of the sector size.
- The *Length* passed to **ZwReadFile** or **ZwWriteFile** must be an integral of the sector size. Note that specifying a read operation to a buffer whose length is exactly the sector size might result in a lesser number of significant bytes being transferred to that buffer if the end of the file was reached during the transfer.
- Buffers must be aligned in accordance with the alignment requirement of the underlying device. To obtain this information, call **ZwCreateFile** to get a handle for the file object that represents the physical device, and pass that handle to ZwQueryInformationFile. For a list of the system's FILE_*XXX*_ALIGNMENT values, see DEVICE_OBJECT.
- Calls to ZwSetInformationFile with the *FileInformationClass* parameter set to **FilePositionInformation** must specify an offset that is a multiple of the sector size.

The FILE_SYNCHRONOUS_IO_ALERT and FILE_SYNCHRONOUS_IO_NONALERT *CreateOptions* flags, which are mutually exclusive as their names suggest, specify that all I/O operations on the file will be synchronous—as long as they occur through the file object referred to by the returned *FileHandle*. All I/O on such a file is serialized across all threads using the returned handle. If either of these *CreateOptions* flags is set, the SYNCHRONIZE *DesiredAccess* flag must also be set—to compel the I/O manager to use the file object as a synchronization object. In these cases, the I/O manager keeps track of the current file-position offset, which you can pass to **ZwReadFile** and **ZwWriteFile**. Call **ZwQueryInformationFile** or **ZwSetInformationFile** to get or set this position.

If the *CreateOptions* FILE_OPEN_REPARSE_POINT flag is not specified and **ZwCreateFile** attempts to open a file with a reparse point, normal reparse point processing occurs for the file. If, on the other hand, the FILE_OPEN_REPARSE_POINT flag is specified, normal reparse processing does not occur and **ZwCreateFile** attempts to directly open the reparse point file. In either case, if the open operation was successful, **ZwCreateFile** returns STATUS_SUCCESS; otherwise, the routine returns an NTSTATUS error code. **ZwCreateFile** never returns STATUS_REPARSE.

The *CreateOptions* FILE_OPEN_REQUIRING_OPLOCK flag eliminates the time between when you open the file and request an oplock that could potentially allow a third party to open the file and get a sharing violation. An application can use the FILE_OPEN_REQUIRING_OPLOCK flag on **ZwCreateFile** and then request any oplock. This ensures that an oplock owner will be notified of any subsequent open request that causes a sharing violation.

In Windows 7, if other handles exist on the file when an application uses the FILE_OPEN_REQUIRING_OPLOCK flag, the create operation will fail with STATUS_OPLOCK_NOT_GRANTED. This restriction no longer exists starting with Windows 8.

If this create operation would break an oplock that already exists on the file, then setting the FILE_OPEN_REQUIRING_OPLOCK flag will cause the create operation to fail with STATUS_CANNOT_BREAK_OPLOCK. The existing oplock will not be broken by this create operation.

An application that uses the FILE_OPEN_REQUIRING_OPLOCK flag must request an oplock after this call succeeds, or all subsequent attempts to open the file will be blocked without the benefit of normal oplock processing. Similarly, if this call succeeds but the subsequent oplock request fails, an application that uses this flag must close its handle after it detects that the oplock request has failed.

> **Note** The FILE_OPEN_REQUIRING_OPLOCK flag is available in Windows 7, Windows Server 2008 R2 and later Windows operating systems. The Microsoft file systems that implement this flag in Windows 7 are NTFS, FAT, and exFAT.

The *CreateOptions* flag FILE_RESERVE_OPFILTER allows an application to request a Level 1, Batch, or Filter oplock to prevent other applications from getting share violations. However, FILE_RESERVE_OPFILTER is only practically useful for Filter oplocks. To use it, you must complete the following steps:

1. Issue a create request with *CreateOptions* of FILE_RESERVE_OPFILTER, *DesiredAccess* of exactly FILE_READ_ATTRIBUTES, and *ShareAccess* of exactly FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE.
    - If there are already open handles, the create request fails with STATUS_OPLOCK_NOT_GRANTED, and the next requested oplock also fails.
    - If you open with more access or less sharing will also cause a failure of STATUS_OPLOCK_NOT_GRANTED.
2. If the create request succeeds, request an oplock.
3. Open another handle to the file to do I/O.

Step three makes this practical only for Filter oplocks. The handle opened in step 3 can have a *DesiredAccess* that contains a maximum of FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | FILE_READ_DATA | FILE_READ_EA | FILE_EXECUTE | SYNCHRONIZE | READ_CONTROL and still not break a Filter oplock. However, any *DesiredAccess* greater than FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES | SYNCHRONIZE will break a Level 1 or Batch oplock and make the FILE_RESERVE_OPFILTER flag useless for those oplock types.

NTFS is the only Microsoft file system that implements FILE_RESERVE_OPFILTER.

Callers of **ZwCreateFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateFile**" instead of "**ZwCreateFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

DEVICE_OBJECT

IO_STATUS_BLOCK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwOpenFile

ZwQueryInformationFile

ZwReadFile

ZwSetInformationFile

ZwWriteFile

Opportunistic Locks

Reparse Points

ZwCreateKey function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateKey** routine creates a new registry key or opens an existing one.

Syntax

```
NTSYSAPI NTSTATUS ZwCreateKey(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              TitleIndex,
  PUNICODE_STRING    Class,
  ULONG              CreateOptions,
  PULONG             Disposition
);
```

Parameters

`KeyHandle`

Pointer to a HANDLE variable that receives a handle to the key.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects (see ACCESS_MASK), the caller can specify one or more of the following access rights, which are specific to object directories:

*DESIREDACCESS*FLAG	ALLOWS CALLER TO DO THIS
KEY_QUERY_VALUE	Read key values.
KEY_SET_VALUE	Write key values.
KEY_CREATE_SUB_KEY	Create subkeys for the key.
KEY_ENUMERATE_SUB_KEYS	Read the key's subkeys.
KEY_CREATE_LINK	Create a symbolic link to the key. This flag is not used by device and intermediate drivers.
KEY_NOTIFY	Ask to receive notification when the name, value, or attributes of the key change. For more information, see ZwNotifyChangeKey.

The caller can also specify one of the following constants, which combines several ACCESS_MASK flags.

CONSTANT	CONSISTS OF THESE ACCESS_MASK FLAGS
KEY_READ	STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY

KEY_WRITE	STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUBKEY
KEY_EXECUTE	Same as KEY_READ.
KEY_ALL_ACCESS	STANDARD_RIGHTS_ALL, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`TitleIndex`

Device and intermediate drivers set this parameter to zero.

`Class`

Pointer to a Unicode string that contains the key's object class. This information is used by the configuration manager.

`CreateOptions`

Specifies the options to apply when creating or opening a key, specified as a compatible combination of the following flags.

***CREATEOPTIONS* FLAG**	**DESCRIPTION**
REG_OPTION_VOLATILE	Key is not preserved when the system is rebooted.
REG_OPTION_NON_VOLATILE	Key is preserved when the system is rebooted.
REG_OPTION_CREATE_LINK	The newly created key is a symbolic link. This flag is not used by device and intermediate drivers.
REG_OPTION_BACKUP_RESTORE	Key should be created or opened with special privileges that allow backup and restore operations. This flag is not used by device and intermediate drivers.

`Disposition`

Pointer to a variable that receives a value indicating whether a new key was created or an existing one opened.

***DISPOSITION* VALUE**	**DESCRIPTION**
REG_CREATED_NEW_KEY	A new key was created.
REG_OPENED_EXISTING_KEY	An existing key was opened.

Return Value

ZwCreateKey returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

ZwCreateKey supplies a handle that the caller can use to manipulate a registry key. For more information, see Using the Registry in a Driver.

Once the handle pointed to by *KeyHandle* is no longer in use, the driver must call ZwClose to close it.

There are two alternate ways to specify the name of the file to be created or opened with **ZwCreateKey**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *ObjectAttributes*. The pathnames of registry keys begin with **\Registry**.
2. As pathname relative to another registry key, represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*.

If the key specified by *ObjectAttributes* does not exist, the routine attempts to create the key. For this attempt to succeed, the new key must be a direct subkey of the key that is referred to by **RootDirectory**, and the key that **RootDirectory** refers to must have been opened for KEY_CREATE_SUB_KEY access.

If the specified key already exists, it is opened and its value is not affected in any way.

The security attributes specified by *ObjectAttributes* when a key is created determine whether the specified *DesiredAccess* is granted on subsequent calls to **ZwCreateKey** and ZwOpenKey.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about working with registry keys, see Using the Registry in a Driver.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateKey**" instead of "**ZwCreateKey**".

The **NtCreateKey** routine in the Windows kernel is not directly accessible to kernel-mode drivers.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport)

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwDeleteKey

ZwEnumerateKey

ZwEnumerateValueKey

ZwFlushKey

ZwNotifyChangeKey

ZwOpenKey

ZwQueryValueKey

ZwSetValueKey

ZwCreateKeyTransacted function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateKeyTransacted** routine creates a new registry key or opens an existing one, and it associates the key with a transaction.

Syntax

```
NTSYSAPI NTSTATUS ZwCreateKeyTransacted(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              TitleIndex,
  PUNICODE_STRING    Class,
  ULONG              CreateOptions,
  HANDLE             TransactionHandle,
  PULONG             Disposition
);
```

Parameters

`KeyHandle`

A pointer to a HANDLE variable into which the routine writes the handle to the key.

`DesiredAccess`

Specifies the type of access to the key that the caller requests. This parameter is an ACCESS_MASK value. For more information, see the description of the *DesiredAccess* parameter of the ZwCreateKey routine.

`ObjectAttributes`

A pointer to the object attributes of the key being opened. This parameter points to an OBJECT_ATTRIBUTES structure that must have been previously initialized by the InitializeObjectAttributes routine. The caller must specify the name of the registry key as the *ObjectName* parameter in the call to **InitializeObjectAttributes**. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`TitleIndex`

Device and intermediate drivers set this parameter to zero.

`Class`

Device and intermediate drivers set this parameter to **NULL**.

`CreateOptions`

Specifies the options to apply when the routine creates or opens the key. Set this parameter to zero or to the bitwise OR of one or more of the following REG_OPTION_*XXX* flag bits.

***CREATEOPTIONS* FLAG**	**DESCRIPTION**
REG_OPTION_VOLATILE	The key is not preserved after the computer restarts.

REG_OPTION_NON_VOLATILE	The key is preserved after the computer restarts.
REG_OPTION_CREATE_LINK	The key is a symbolic link. This flag is not used by device and intermediate drivers.
REG_OPTION_BACKUP_RESTORE	Open the key with special privileges that enable backup and restore operations. This flag is not used by device and intermediate drivers.

`TransactionHandle`

A handle to a transaction object. To obtain this handle, you can call the ZwCreateTransaction routine. Or, if you have a pointer to a transaction object, you can supply the pointer to the ObOpenObjectByPointer routine to obtain the corresponding transaction handle.

`Disposition`

A pointer to a location into which the routine writes one of the following values to indicate whether the call created a new key or opened an existing one.

***DISPOSITION* VALUE**	**DESCRIPTION**
REG_CREATED_NEW_KEY	A new key was created.
REG_OPENED_EXISTING_KEY	An existing key was opened.

You can set *Disposition* = **NULL** if this information is not needed.

Return Value

ZwCreateKeyTransacted returns STATUS_SUCCESS if the call successfully creates or opens the key. Possible error return values include the following:

RETURN CODE	**DESCRIPTION**
STATUS_INVALID_PARAMETER	The *ObjectAttributes* parameter is **NULL** or points to invalid information, or the *CreateOptions* parameter value specifies invalid options.

STA TUS _OB JEC T_P AT H_S YNT AX_ BA D	The registry path in the object attributes is invalid.
STA TUS _OB JEC T_N AM E_N OT_ FOU ND	The registry path in the object attributes was not found.
STA TUS _AC CES S_D ENI ED	The caller did not have the required access rights to open a handle for the named registry key.
STA TUS _IN SUF FICI ENT _RE SO URC ES	A memory allocation operation failed.

Remarks

This routine provides a handle that the caller can access a registry key with. Additionally, this routine associates the key with an active transaction.

After the handle that is pointed to by *KeyHandle* is no longer being used, the driver must call the ZwClose routine to close it.

Like **ZwCreateKeyTransacted**, the ZwOpenKeyTransacted routine associates a key with a transaction. Unlike **ZwCreateKeyTransacted**, which can create a new key or open an existing key, **ZwOpenKeyTransacted** can only open a registry key that already exists.

After a kernel-mode driver obtains a handle to a transaction (for example, by calling **ZwCreateTransaction**), the

driver can perform a series of registry operations that are part of this transaction. The driver can close the transaction either by committing to the changes that were made in the transaction or by rolling back the transaction.

After the driver successfully completes all registry operations that are part of a transaction, it can call the ZwCommitTransaction routine to commit to the changes. The driver can call the ZwRollbackTransaction routine to roll back the transaction.

During a transaction, a registry operation is part of the transaction if the system call that performs the operation meets either of the following conditions:

- The call specifies, as an input parameter, the transaction handle. For example, calls to **ZwCreateKeyTransacted** and **ZwOpenKeyTransacted** can associate one or more handles to registry keys with the transaction.
- The call specifies, as an input parameter, a registry key handle that was obtained by a call to **ZwCreateKeyTransacted** or **ZwOpenKeyTransacted** to which the transaction handle was supplied. For example, a call to the ZwSetValueKey routine can use a key handle that was obtained in this way to set the value of a registry key as part of a transaction.

For more information about kernel-mode transactions, see Using Kernel Transaction Manager.

The security descriptor in the object attributes determines whether the access rights that are specified by the *DesiredAccess* parameter are granted on later calls to routines, such as **ZwOpenKeyTransacted** that access the key, or to routines, such as **ZwCreateKeyTransacted,** that create subkeys of the key.

If the kernel-mode caller is not running in a system thread context, it must ensure that any handles it creates are kernel handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about how to work with registry keys in kernel mode, see Using the Registry in a Driver.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

ObOpenObjectByPointer

ZwClose

ZwCommitTransaction

ZwCreateKey

ZwCreateTransaction

ZwOpenKeyTransacted

ZwRollbackTransaction

ZwSetValueKey

ZwCreateResourceManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwCreateResourceManager** routine creates a resource manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCreateResourceManager(
  PHANDLE            ResourceManagerHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             TmHandle,
  LPGUID             ResourceManagerGuid,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              CreateOptions,
  PUNICODE_STRING    Description
);
```

Parameters

`ResourceManagerHandle`

A pointer to a caller-allocated variable that receives a handle to the new resource manager object if the call to **ZwCreateResourceManager** is successful.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the resource manager object. In addition to the access rights that are defined for all kinds of objects (see **ACCESS_MASK**), the caller can specify any of the following access right flags for resource manager objects:

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
RESOURCEMANAGER_ENLIST	Enlist in transactions (see ZwCreateEnlistment).
RESOURCEMANAGER_GET_NOTIFICATION	Receive notifications about the transactions that are associated with this resource manager (see ZwGetNotificationResourceManager).
RESOURCEMANAGER_REGISTER_PROTOCOL	Not used.
RESOURCEMANAGER_QUERY_INFORMATION	Query information about the resource manager (see ZwQueryInformationResourceManager).
RESOURCEMANAGER_SET_INFORMATION	Not used.
RESOURCEMANAGER_RECOVER	Recover the resource manager (see ZwRecoverResourceManager).
RESOURCEMANAGER_COMPLETE_PROPAGATION	Not used.

Alternatively, you can specify one or more of the following generic ACCESS_MASK flags. (The STANDARD_RIGHTS_*Xxx* flags are predefined system values that are used to enforce security on system objects.)

You can also combine these generic flags with additional flags from the preceding table. The following table shows how generic access rights correspond to specific access rights.

GENERIC ACCESS RIGHT	SET OF SPECIFIC ACCESS RIGHTS
RESOURCEMANAGER_GENERIC_READ	STANDARD_RIGHTS_READ, RESOURCEMANAGER_QUERY_INFORMATION, and SYNCHRONIZE
RESOURCEMANAGER_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, RESOURCEMANAGER_SET_INFORMATION, RESOURCEMANAGER_RECOVER, RESOURCEMANAGER_ENLIST, RESOURCEMANAGER_GET_NOTIFICATION, RESOURCEMANAGER_REGISTER_PROTOCOL, RESOURCEMANAGER_COMPLETE_PROPAGATION, and SYNCHRONIZE
RESOURCEMANAGER_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, RESOURCEMANAGER_RECOVER, RESOURCEMANAGER_ENLIST, RESOURCEMANAGER_GET_NOTIFICATION, RESOURCEMANAGER_COMPLETE_PROPAGATION, and SYNCHRONIZE
RESOURCEMANAGER_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, RESOURCEMANAGER_GENERIC_READ, RESOURCEMANAGER_GENERIC_WRITE, and RESOURCEMANAGER_GENERIC_EXECUTE

`TmHandle`

A handle to a transaction manager object that was obtained by a previous all to ZwCreateTransactionManager or ZwOpenTransactionManager.

`ResourceManagerGuid`

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

CREATEOPTIONS FLAG	MEANING
RESOURCE_MANAGER_COMMUNICATION	For internal use only.
RESOURCE_MANAGER_VOLATILE	The caller will manage volatile resources. It will be non-persistent and will not perform recovery.

This parameter is optional and can be zero.

`Description`

A pointer to a caller-supplied UNICODE_STRING structure that contains a NULL-terminated string. The string

provides a description of the resource manager. KTM stores a copy of the string and includes the string in messages that it writes to the log stream. The maximum string length is MAX_RESOURCEMANAGER_DESCRIPTION_LENGTH. This parameter is optional and can be **NULL**.

Return Value

ZwCreateResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that *TmHandle* specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The handle that *TmHandle* specifies is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the specified transaction manager object.
STATUS_TRANSACTION_OBJECT_EXPIRED	The handle that *TmHandle* specifies is closed.

STATUS_INVALID_PARAMETER	The *CreateOptions*parameter's value is invalid or the *Description* parameter's string is too long.
STATUS_TM_VOLATILE	The *CreateOptions*parameter does not specify RESOURCE_MANAGER_VOLATILE but the transaction manager that *TmHandle* specifies is volatile.
STATUS_OBJECT_NAME_COLLISION	The GUID that *ResourceManagerGuid*specifies already exists.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

A resource manager that calls **ZwCreateResourceManager** must eventually call ZwClose to close the object handle.

For more information about **ZwCreateResourceManager**, see Creating a Resource Manager.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

ZwClose

ZwCreateEnlistment

ZwCreateTransactionManager

ZwGetNotificationResourceManager

ZwOpenResourceManager

ZwOpenTransactionManager

ZwQueryInformationResourceManager

ZwRecoverResourceManager

ZwCreateSection function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwCreateSection** routine creates a section object.

Syntax

```
NTSYSAPI NTSTATUS ZwCreateSection(
  PHANDLE            SectionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PLARGE_INTEGER     MaximumSize,
  ULONG              SectionPageProtection,
  ULONG              AllocationAttributes,
  HANDLE             FileHandle
);
```

Parameters

`SectionHandle`

Pointer to a HANDLE variable that receives a handle to the section object.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. In addition to the access rights that are defined for all types of objects (see ACCESS_MASK), the caller can specify any of the following access rights, which are specific to section objects:

***DESIREDACCESS* FLAG**	**ALLOWS CALLER TO DO THIS**
SECTION_EXTEND_SIZE	Dynamically extend the size of the section.
SECTION_MAP_EXECUTE	Execute views of the section.
SECTION_MAP_READ	Read views of the section.
SECTION_MAP_WRITE	Write views of the section.
SECTION_QUERY	Query the section object for information about the section. Drivers should set this flag.
SECTION_ALL_ACCESS	All of the previous flags combined with STANDARD_RIGHTS_REQUIRED.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`MaximumSize`

Specifies the maximum size, in bytes, of the section. **ZwCreateSection** rounds this value up to the nearest multiple of PAGE_SIZE. If the section is backed by the paging file, *MaximumSize* specifies the actual size of the section. If the section is backed by an ordinary file, *MaximumSize* specifies the maximum size that the file can be extended or mapped to.

`SectionPageProtection`

Specifies the protection to place on each page in the section. Use one of the following four values: PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, or PAGE_WRITECOPY. For a description of these values, see CreateFileMapping.

`AllocationAttributes`

Specifies a bitmask of SEC_*XXX* flags that determines the allocation attributes of the section. For a description of these flags, see CreateFileMapping.

`FileHandle`

Optionally specifies a handle for an open file object. If the value of *FileHandle* is **NULL**, the section is backed by the paging file. Otherwise, the section is backed by the specified file.

Return Value

ZwCreateSection returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_FILE_LOCK_CONFLICT	The file specified by the *FileHandle* parameter is locked.
STATUS_INVALID_FILE_FOR_SECTION	The file specified by *FileHandle* does not support sections.

STA TUS _IN VAL ID_ PA GE_ PR OTE CTI ON	The value specified for the *SectionPageProtection* parameter is invalid.
STA TUS _M APP ED_ FILE _SIZ E_Z ERO	The size of the file specified by *FileHandle* is zero, and *MaximumSize* is zero.
STA TUS _SE CTI ON_ TO O_B IG	The value of *MaximumSize* is too big. This occurs when either *MaximumSize* is greater than the system-defined maximum for sections, or if *MaximumSize* is greater than the specified file and the section is not writable.

Remarks

Once the handle pointed to by *SectionHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about setting up mapped sections and views of memory, see Sections and Views.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtCreateSection**" instead of "**ZwCreateSection**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

ACCESS_MASK

CreateFileMapping

InitializeObjectAttributes

ZwClose

ZwMapViewOfSection

ZwOpenSection

ZwUnmapViewOfSection

ZwCreateTransaction function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwCreateTransaction** routine creates a transaction object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCreateTransaction(
  PHANDLE            TransactionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  LPGUID             Uow,
  HANDLE             TmHandle,
  ULONG              CreateOptions,
  ULONG              IsolationLevel,
  ULONG              IsolationFlags,
  PLARGE_INTEGER     Timeout,
  PUNICODE_STRING    Description
);
```

Parameters

`TransactionHandle`

A pointer to a caller-allocated variable that receives a handle to the new transaction object, if the call to **ZwCreateTransaction** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following flags for transaction objects.

ACCESS MASK	ALLOWS THE CALLER TO
TRANSACTION_COMMIT	Commit the transaction (see ZwCommitTransaction).
TRANSACTION_ENLIST	Create an enlistment for the transaction (see ZwCreateEnlistment).
TRANSACTION_PROPAGATE	Do not use.
TRANSACTION_QUERY_INFORMATION	Obtain information about the transaction (see ZwQueryInformationTransaction).
TRANSACTION_ROLLBACK	Roll back the transaction (see ZwRollbackTransaction).
TRANSACTION_SET_INFORMATION	Set information for the transaction (see ZwSetInformationTransaction).

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the ACCESS_MASK reference page. You can also combine these bitmaps with additional flags from the preceding table. The following

table shows how the bitmaps correspond to specific access rights.

RIGHTS BITMAP	SET OF SPECIFIC ACCESS RIGHTS
TRANSACTION_GENERIC_READ	STANDARD_RIGHTS_READ, TRANSACTION_QUERY_INFORMATION, and SYNCHRONIZE
TRANSACTION_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, TRANSACTION_SET_INFORMATION, TRANSACTION_COMMIT, TRANSACTION_ENLIST, TRANSACTION_ROLLBACK, TRANSACTION_PROPAGATE, TRANSACTION_SAVEPOINT, and SYNCHRONIZE
TRANSACTION_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, TRANSACTION_COMMIT, TRANSACTION_ROLLBACK, and SYNCHRONIZE
TRANSACTION_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, TRANSACTION_GENERIC_READ, TRANSACTION_GENERIC_WRITE, and TRANSACTION_GENERIC_EXECUTE
TRANSACTION_RESOURCE_MANAGER_RIGHTS	STANDARD_RIGHTS_WRITE, TRANSACTION_GENERIC_READ, TRANSACTION_SET_INFORMATION, TRANSACTION_ENLIST, TRANSACTION_ROLLBACK, TRANSACTION_PROPAGATE, and SYNCHRONIZE

Typically, a resource manager specifies TRANSACTION_RESOURCE_MANAGER_RIGHTS.

The *DesiredAccess* value cannot be zero.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`Uow`

A pointer to a GUID that KTM uses as the new transaction object's unit of work (UOW) identifier. This parameter is optional and can be **NULL**. If this parameter is **NULL**, KTM generates a GUID and assigns it to the transaction object. For more information, see the following Remarks section.

`TmHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. KTM assigns the new transaction object to the specified transaction manager object. If this parameter is **NULL**, KTM assigns the new transaction object to a transaction manager later, when a resource manager creates an enlistment for the transaction.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

OPTION FLAG	MEANING
TRANSACTION_DO_NOT_PROMOTE	Reserved for future use.

`IsolationLevel`

Reserved for future use. Callers must set this parameter to zero.

`IsolationFlags`

Reserved for future use. Callers should set this parameter to zero.

`Timeout`

A pointer to a time-out value. If the transaction has not been committed by the time specified by this parameter, KTM rolls back the transaction. The time-out value is expressed in system time units (100-nanosecond intervals), and can specify either an absolute time or a relative time. If the value pointed to by *Timeout* is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. This pointer is optional and can be **NULL** if you do not want the transaction to have a time-out value. If *Timeout* = **NULL** or * *Timeout* = 0, the transaction never times out. (You can also use ZwSetInformationTransaction to set a time-out value.)

`Description`

A pointer to a caller-supplied UNICODE_STRING structure that contains a NULL-terminated string. The string provides a description of the transaction. KTM stores a copy of the string and includes the string in messages that it writes to the log stream. The maximum string length is MAX_TRANSACTION_DESCRIPTION_LENGTH. This parameter is optional and can be **NULL**.

Return Value

ZwCreateTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *CreateOptions* parameter contains an invalid flag, the *DesiredAccess* parameter is zero, or the *Description* parameter's string is too long.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).

STA TUS _IN VAL ID_ ACL	A security descriptor contains an invalid access control list (ACL).
STA TUS _IN VAL ID_S ID	A security descriptor contains an invalid security identifier (SID).
STA TUS _OB JEC T_N AM E_E XIS TS	The object name that the *ObjectAttributes* parameter specifies already exists.
STA TUS _OB JEC T_N AM E_IN VAL ID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STA TUS _AC CES S_D ENI ED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

The caller can use the *Uow* parameter to specify a UOW identifier for the transaction object. If the caller does not specify a UOW identifier, KTM generates a GUID and assigns it to the transaction object. The caller can later obtain this GUID by calling ZwQueryInformationTransaction.

Typically, you should let KTM generate a GUID for the transaction object, unless your component communicates with another TPS component that has already generated a UOW identifier for the transaction.

To close the transaction handle, the component that called **ZwCreateTransaction** must call ZwClose. If the last transaction handle closes before any component calls ZwCommitTransaction for the transaction, KTM rolls back the transaction.

For more information about how transaction clients should use **ZwCreateTransaction**, see Creating a Transactional Client.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCommitTransaction

ZwCreateEnlistment

ZwCreateTransactionManager

ZwOpenTransaction

ZwOpenTransactionManager

ZwQueryInformationTransaction

ZwRollbackTransaction

ZwSetInformationTransaction

ZwCreateTransactionManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwCreateTransactionManager** routine creates a new transaction manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwCreateTransactionManager(
  PHANDLE            TmHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PUNICODE_STRING    LogFileName,
  ULONG              CreateOptions,
  ULONG              CommitStrength
);
```

Parameters

`TmHandle`

A pointer to a caller-allocated variable that receives a handle to the new transaction manager object.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction manager object. In addition to the access rights that are defined for all kinds of objects (see ACCESS_MASK), the caller can specify any of the following access right flags for transaction manager objects.

ACCESS_MASK FLAG	ALLOWS THE CALLER TO
TRANSACTIONMANAGER_CREATE_RM	Create a resource manager (see ZwCreateResourceManager).
TRANSACTIONMANAGER_QUERY_INFORMATION	Obtain information about the transaction manager (see ZwQueryInformationTransactionManager and ZwEnumerateTransactionObject). Also required for ZwOpenResourceManager, ZwCreateTransaction, and ZwOpenTransaction.)
TRANSACTIONMANAGER_RECOVER	Recover the transaction manager (see ZwRecoverTransactionManager and ZwRollforwardTransactionManager).
TRANSACTIONMANAGER_RENAME	Not used.
TRANSACTIONMANAGER_SET_INFORMATION	Not used.

Alternatively, you can specify one or more of the following ACCESS_MASK bitmaps. These bitmaps combine the flags from the previous table with the STANDARD_RIGHTS_*XXX* flags that are described on the **ACCESS_MASK** reference page. You can also combine these bitmaps with additional flags from the preceding table. The following table shows how the bitmaps correspond to specific access rights.

RIGHTS BITMAP	SET OF SPECIFIC ACCESS RIGHTS

TRANSACTIONMANAGER_GENERIC_READ	STANDARD_RIGHTS_READ and TRANSACTIONMANAGER_QUERY_INFORMATION
TRANSACTIONMANAGER_GENERIC_WRITE	STANDARD_RIGHTS_WRITE, TRANSACTIONMANAGER_SET_INFORMATION, TRANSACTIONMANAGER_RECOVER, TRANSACTIONMANAGER_RENAME, and TRANSACTIONMANAGER_CREATE_RM
TRANSACTIONMANAGER_GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE
TRANSACTIONMANAGER_ALL_ACCESS	STANDARD_RIGHTS_REQUIRED, TRANSACTIONMANAGER_GENERIC_READ, TRANSACTIONMANAGER_GENERIC_WRITE, and TRANSACTIONMANAGER_GENERIC_EXECUTE

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of a CLFS log file stream to be associated with the transaction manager object. This parameter must be **NULL** if the *CreateOptions* parameter is TRANSACTION_MANAGER_VOLATILE. Otherwise, this parameter must be non-**NULL**. For more information, see the following Remarks section.

`CreateOptions`

Optional object creation flags. The following table contains the available flags, which are defined in Ktmtypes.h.

OPTION FLAG	MEANING
TRANSACTION_MANAGER_VOLATILE	The transaction manager object will be volatile. Therefore, it will not use a log file.
TRANSACTION_MANAGER_COMMIT_DEFAULT	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_VOLUME	For internal use only.
TRANSACTION_MANAGER_COMMIT_SYSTEM_HIVES	For internal use only.
TRANSACTION_MANAGER_COMMIT_LOWEST	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_RECOVERY	For internal use only.
TRANSACTION_MANAGER_CORRUPT_FOR_PROGRESS	For internal use only.

`CommitStrength`

Reserved for future use. This parameter must be zero.

Return Value

ZwCreateTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of an input parameter is invalid.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).
STATUS_LOG_CORRUPTION_DETECTED	KTM encountered an error while creating or opening the log file.
STATUS_INVALID_ACL	A security descriptor contains an invalid access control list (ACL).
STATUS_INVALID_SID	A security descriptor contains an invalid security identifier (SID).

STATUS_OBJECT_NAME_EXISTS	The object name that the *ObjectAttributes* parameter specifies already exists.
STATUS_OBJECT_NAME_COLLISION	The operating system detected a duplicate object name. The error might indicate that the log stream is already being used.
STATUS_OBJECT_NAME_INVALID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

If the log file stream that the *LogFileName* parameter specifies does not exist, KTM calls CLFS to create the stream. If the stream already exists, KTM calls CLFS to open the stream.

Your TPS component must call ZwRecoverTransactionManager after it has called **ZwCreateTransactionManager**

If your TPS component specifies the TRANSACTION_MANAGER_VOLATILE flag in the *CreateOptions* parameter, all resource managers that are associated with the transaction manager object must specify the RESOURCE_MANAGER_VOLATILE flag when they call ZwCreateResourceManager.

A TPS component that calls **ZwCreateTransactionManager** must eventually call ZwClose to close the object handle.

For more information about how use **ZwCreateTransactionManager**, see Creating a Resource Manager.

NtCreateTransactionManager and **ZwCreateTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwOpenTransactionManager

ZwQueryInformationTransactionManager

ZwRecoverTransactionManager

ZwRollforwardTransactionManager

ZwDeleteKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwDeleteKey** routine deletes an open key from the registry.

Syntax

```
NTSYSAPI NTSTATUS ZwDeleteKey(
  HANDLE KeyHandle
);
```

Parameters

`KeyHandle`

Handle to the registry key to be deleted. The handle is created by a successful call to ZwCreateKey or ZwOpenKey.

Return Value

ZwDeleteKey returns an NTSTATUS value. Possible return values include:

- **STATUS_SUCCESS**
- **STATUS_ACCESS_DENIED**
- **STATUS_INVALID_HANDLE**

Remarks

The handle must have been opened for DELETE access for this routine to succeed. For more information, see the *DesiredAccess* parameter for ZwCreateKey.

A call to **ZwDeleteKey** causes the handle that is specified in the *KeyHandle* parameter—and all other handles to the deleted key—to become invalid. After a call to **ZwDeleteKey** invalidates the key handles, you must call ZwClose to close the key handles.

For more information about working with registry keys, see Using the Registry in a Driver.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtDeleteKey**" instead of "**ZwDeleteKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateKey

ZwOpenKey

ZwDeleteValueKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwDeleteValueKey** routine deletes a value entry matching a name from an open key in the registry. If no such entry exists, an error is returned.

Syntax

```
NTSYSAPI NTSTATUS ZwDeleteValueKey(
  HANDLE          KeyHandle,
  PUNICODE_STRING ValueName
);
```

Parameters

`KeyHandle`

The handle to the registry key containing the value entry of interest. This key must have been opened with KEY_SET_VALUE set for the desired access. This handle is created by a successful call to ZwCreateKey or ZwOpenKey.

`ValueName`

Pointer to a UNICODE_STRING structure that contains the name of the value entry to delete. This parameter can be an empty string if the value entry has no name.

Return Value

ZwDeleteValueKey returns STATUS_SUCCESS or an appropriate error status representing the final completion status of the operation. Possible error status codes include the following:

RETURN CODE	DESCRIPTION
STATUS_ACCESS_DENIED	The *KeyHandle* handle does not have the KEY_SET_VALUE access.

STATUS_INSUFFICIENT_RESOURCES	Additional resources required by this function were not available.
STATUS_INVALID_HANDLE	The specified *KeyHandle* parameter was a **NULL** pointer or not a valid pointer to an open registry key.
STATUS_OBJECT_NAME_NOT_FOUND	The *ValueName* registry key entry was not found.

Remarks

The *KeyHandle* passed to **ZwDeleteValueKey** must have been opened for delete access to succeed. The *DesiredAccess* values of KEY_SET_VALUE, KEY_WRITE, and KEY_ALL_ACCESS include the KEY_SET_VALUE access mask required for delete access. For a description of possible values for *DesiredAccess*, see ZwCreateKey.

If callback functions are registered for this registry key, then these callback functions will be called.

Device drivers should not attempt to call **ZwDeleteValueKey** directly to delete value entries in a subkey of the **\Registry..\ResourceMap** key. Only the system can write or delete value entries in the **\Registry..\HardwareDescription** tree.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtDeleteValueKey**" instead of "**ZwDeleteValueKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateKey

ZwDeleteKey

ZwEnumerateValueKey

ZwOpenKey

ZwQueryValueKey

ZwSetValueKey

ZwEnumerateKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwEnumerateKey** routine returns information about a subkey of an open registry key.

Syntax

```
NTSYSAPI NTSTATUS ZwEnumerateKey(
  HANDLE                KeyHandle,
  ULONG                 Index,
  KEY_INFORMATION_CLASS KeyInformationClass,
  PVOID                 KeyInformation,
  ULONG                 Length,
  PULONG                ResultLength
);
```

Parameters

`KeyHandle`

Handle to the registry key that contains the subkeys to be enumerated. The handle is created by a successful call to ZwCreateKey or ZwOpenKey.

`Index`

The index of the subkey that you want information for. If the key has *n* subkeys, the subkeys are numbered from 0 to *n*-1.

`KeyInformationClass`

Specifies a KEY_INFORMATION_CLASS enumeration value that determines the type of information to be received by the *KeyInformation* buffer. Set *KeyInformationClass* to one of the following values:

- **KeyBasicInformation**
- **KeyFullInformation**
- **KeyNodeInformation**

If any value not in this list is specified, the routine returns error code STATUS_INVALID_PARAMETER.

`KeyInformation`

Pointer to a caller-allocated buffer that receives the requested information. The *KeyInformationClass* parameter determines the type of information provided.

`Length`

Specifies the size, in bytes, of the *KeyInformation* buffer.

`ResultLength`

Pointer to a variable that receives the size, in bytes, of the registry-key information. If **ZwEnumerateKey** returns STATUS_SUCCESS, you can use the value of this variable to determine the amount of data returned. If the routine returns STATUS_BUFFER_OVERFLOW or STATUS_BUFFER_TOO_SMALL, you can use the value of this variable to determine the size of buffer required to hold the key information.

Return Value

ZwEnumerateKey returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. Possible error code values include:

RETURN CODE	DESCRIPTION
STATUS_BUFFER_OVERFLOW	The buffer supplied is too small, and only partial data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_BUFFER_TOO_SMALL	The buffer supplied is too small, and no data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_INVALID_PARAMETER	The *KeyInformationClass* parameter is not a valid KEY_INFORMATION_CLASS value.
STATUS_NO_MORE_ENTRIES	The *Index* value is out of range for the registry key specified by *KeyHandle*. For example, if a key has *n* subkeys, then for any value greater than *n*-1 the routine returns STATUS_NO_MORE_ENTRIES.

Remarks

The handle must have been opened with KEY_ENUMERATE_SUB_KEYS access. This is accomplished by passing KEY_ENUMERATE_SUB_KEYS, KEY_READ, or KEY_ALL_ACCESS as the *DesiredAccess* parameter to ZwCreateKey or ZwOpenKey.

The *Index* parameter is simply a way to select among subkeys of the key referred to by the *KeyHandle*. Two calls to **ZwEnumerateKey** with the same *Index* are not guaranteed to return the same result.

For more information about working with registry keys, see Using the Registry in a Driver.

Note If the call to this function occurs in user mode, you should use the name "**NtEnumerateKey**" instead of "**ZwEnumerateKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

KEY_BASIC_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NODE_INFORMATION

RtlCheckRegistryKey

RtlCreateRegistryKey

RtlDeleteRegistryValue

RtlQueryRegistryValues

RtlWriteRegistryValue

ZwCreateKey

ZwEnumerateValueKey

ZwOpenKey

ZwEnumerateTransactionObject function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwEnumerateTransactionObject** routine enumerates the KTM objects on a computer.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwEnumerateTransactionObject(
  HANDLE            RootObjectHandle,
  KTMOBJECT_TYPE    QueryType,
  PKTMOBJECT_CURSOR ObjectCursor,
  ULONG             ObjectCursorLength,
  PULONG            ReturnLength
);
```

Parameters

`RootObjectHandle`

A handle to a KTM object. The routine enumerates the child objects of the specified object. This parameter is optional and can be **NULL**. For more information about valid values for this parameter, see the table in the following Remarks section.

`QueryType`

A KTMOBJECT_TYPE-typed value that identifies the object type to enumerate. For more information about valid values for this parameter, see the table in the following Remarks section.

`ObjectCursor`

A pointer to a caller-allocated buffer that begins with a KTMOBJECT_CURSOR structure. **ZwEnumerateTransactionObject** uses the buffer to store the GUIDs of objects that it finds.

`ObjectCursorLength`

The length, in bytes, of the buffer that *ObjectCursor* points to.

`ReturnLength`

A pointer to a caller-allocated location that receives the number of bytes that **ZwEnumerateTransactionObject** returns in the *ObjectCursor* buffer, including the length of the **KTMOBJECT_CURSOR** structure and the length of all returned GUIDs.

Return Value

ZwEnumerateTransactionObject returns STATUS_SUCCESS if the operation succeeds but the routine has not enumerated all the objects. If there are no more objects to enumerate, the routine returns STATUS_NO_MORE_ENTRIES. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The *QueryType* or *ObjectCursorLength* parameter's value is invalid.
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *RootObjectHandle* parameter specifies is not a handle to a valid KTM object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the objects that are being enumerated.

The routine might return other NTSTATUS values.

Remarks

The following table contains the valid values for the *RootObjectHandle* and *QueryType* parameters.

***QUERYTYPE* PARAMETER**	***ROOTOBJECTHANDLE* PARAMETER**	**OBJECTS ENUMERATED**
KTMOBJECT_TRANSACTION_MANAGER	**NULL**	All transaction manager objects
KTMOBJECT_RESOURCE_MANAGER	A handle to a transaction manager object. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.	All resource manager objects that belong to the specified transaction manager object

KTMOBJECT_ENLISTMENT	A handle to a resource manager object. The handle must have RESOURCEMANAGER_QUERY_INFORMATION access to the object.	All enlistment objects that belong to the specified resource manager object
KTMOBJECT_TRANSACTION	A handle to a transaction manager object. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.	All transaction objects that belong to the specified transaction manager object
KTMOBJECT_TRANSACTION	**NULL**	All transaction objects that belong to all transaction manager objects

Most TPS components do not have to call **ZwEnumerateTransactionObject**, but the routine might be useful if you have to write a debugging utility.

Before your component calls **ZwEnumerateTransactionObject**, it must allocate and zero the buffer that *ObjectCursor* points to. The buffer's GUID array can be large enough to receive one or more elements.

To enumerate all of the KTM objects of the specified type, your component must call **ZwEnumerateTransactionObject** repeatedly until it returns STATUS_NO_MORE_ENTRIES.

Every time that the routine is called, it fills the buffer's GUID array with as many object GUIDs that will fit. After each call, your component can use the KTMOBJECT_CURSOR structure's **ObjectIdCount** member to determine the number of object GUIDs that the routine stored in the array.

NtEnumerateTransactionObject and **ZwEnumerateTransactionObject** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Examples

The following code example shows how to enumerate all of the transaction objects on a computer. In this example, the **KTMOBJECT_CURSOR** structure's GUID array contains only one element, so each call to **ZwEnumerateTransactionObject** returns one GUID. The routine creates a Unicode string from the GUID and displays the string.

```
NTSTATUS Status;
UNICODE_STRING GuidString;
KTMOBJECT_CURSOR Cursor;
ULONG ReturnedBytes;

RtlZeroMemory(&Cursor, sizeof(Cursor));

do {
    Status = ZwEnumerateTransactionObject(
                                          NULL,
                                          KTMOBJECT_TRANSACTION,
                                          &Cursor,
                                          sizeof(Cursor),
                                          &ReturnedBytes
                                          );

    if (Status != STATUS_NO_MORE_ENTRIES) {
        RtlStringFromGUID(
                          &Cursor.ObjectIds[0],
                          &GuidString
                          );
        OutputMessage(GuidString.Buffer);
        OutputMessage(L"\r\n");
        RtlFreeUnicodeString(&GuidString);
    }
} while (Status == STATUS_SUCCESS);
if (Status == STATUS_NO_MORE_ENTRIES) {
    Status = STATUS_SUCCESS;
}
```

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KTMOBJECT_CURSOR

KTMOBJECT_TYPE

Using Nt and Zw Versions of the Native System Services Routines

ZwEnumerateValueKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwEnumerateValueKey** routine gets information about the value entries of an open key.

Syntax

```
NTSYSAPI NTSTATUS ZwEnumerateValueKey(
  HANDLE                      KeyHandle,
  ULONG                       Index,
  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
  PVOID                       KeyValueInformation,
  ULONG                       Length,
  PULONG                      ResultLength
);
```

Parameters

`KeyHandle`

Handle to the registry key that you want to enumerate value entries for. A successful call to ZwCreateKey or ZwOpenKey creates this handle.

`Index`

The zero-based index of the subkey that you want value information for.

`KeyValueInformationClass`

Specifies a KEY_VALUE_INFORMATION_CLASS value that determines the type of information returned in the *KeyValueInformation* buffer.

`KeyValueInformation`

Pointer to a caller-allocated buffer that receives the requested information.

`Length`

Specifies the size, in bytes, of the *KeyValueInformation* buffer.

`ResultLength`

Pointer to a variable that receives the size, in bytes, of the value information. If this routine returns STATUS_SUCCESS, the variable indicates the amount of data returned. If this routine returns STATUS_BUFFER_OVERFLOW or STATUS_BUFFER_TOO_SMALL, the variable indicates the buffer size that is required to hold the value information.

Return Value

ZwEnumerateValueKey returns STATUS_SUCCESS on success, or the appropriate error code on failure. Possible error code values include:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_OVERFLOW	The buffer supplied is too small, and only partial data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_BUFFER_TOO_SMALL	The buffer supplied is too small, and no data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_INVALID_PARAMETER	The *KeyInformationClass* parameter is not a valid KEY_VALUE_INFORMATION_CLASS value.
STATUS_NO_MORE_ENTRIES	The *Index* value is out of range for the registry key specified by *KeyHandle*. For example, if a key has *n* subkeys, then for any value greater than *n*-1 the routine returns STATUS_NO_MORE_ENTRIES.

Remarks

The*KeyHandle* passed to **ZwEnumerateValueKey** must have been opened with KEY_QUERY_VALUE access. This is accomplished by passing KEY_QUERY_VALUE, KEY_READ, or KEY_ALL_ACCESS as the *DesiredAccess* parameter to ZwCreateKey or ZwOpenKey.

The *Index* is simply a way to select among subkeys with value entries. Two calls to **ZwEnumerateValueKey** with the same *Index* are not guaranteed to return the same results.

For more information about working with registry keys, see Using the Registry in a Driver.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtEnumerateValueKey**" instead of "**ZwEnumerateValueKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

KEY_VALUE_BASIC_INFORMATION

KEY_VALUE_FULL_INFORMATION

KEY_VALUE_PARTIAL_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateKey

ZwOpenKey

ZwQueryValueKey

ZwFlushKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwFlushKey** routine forces a registry key to be committed to disk.

Syntax

```
NTSYSAPI NTSTATUS ZwFlushKey(
  HANDLE KeyHandle
);
```

Parameters

`KeyHandle`

Handle to the registry key to be flushed to disk. This handle is created by a successful call to ZwCreateKey or ZwOpenKey.

Return Value

ZwFlushKey returns STATUS_SUCCESS if the key information was transferred to disk, or the appropriate error code on failure.

Remarks

You can flush changes made by ZwCreateKey or ZwSetValueKey by calling **ZwFlushKey**. This routine does not return until all the changed data that is associated with *KeyHandle* has been written to disk. **ZwFlushKey** flushes the entire registry hive for the key, which includes every subkey of the key specified.

> **Note** This routine can flush the entire registry. Accordingly, it can generate a great deal of I/O. Since the system flushes key changes automatically every few seconds, you seldom need to call **ZwFlushKey**.

For more information about working with registry keys, see Using the Registry in a Driver.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtFlushKey**" instead of "**ZwFlushKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateKey

ZwOpenKey

ZwSetValueKey

ZwGetNotificationResourceManager function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwGetNotificationResourceManager** routine retrieves the next transaction notification from a specified resource manager's notification queue.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwGetNotificationResourceManager(
  HANDLE                    ResourceManagerHandle,
  PTRANSACTION_NOTIFICATION TransactionNotification,
  ULONG                     NotificationLength,
  PLARGE_INTEGER            Timeout,
  PULONG                    ReturnLength,
  ULONG                     Asynchronous,
  ULONG_PTR                 AsynchronousContext
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_GET_NOTIFICATION access to the object.

`TransactionNotification`

A pointer to a caller-allocated buffer that receives information about the retrieved notification. The buffer must be large enough to contain a TRANSACTION_NOTIFICATION structure plus additional notification-specific arguments.

`NotificationLength`

The length, in bytes, of the buffer that the *TransactionNotification*parameter points to.

`Timeout`

A pointer to a value that specifies a relative or absolute time, in units of 100 nanoseconds. This pointer is optional and can be **NULL**.

If the pointer is **NULL**, **ZwGetNotificationResourceManager** does not return until a transaction notification is available. If a time value is specified, **ZwGetNotificationResourceManager** returns when a notification is available or after the specified time elapses, whichever comes first.

A negative value specifies a time that is relative to the current system time. For example, a relative time value of five seconds causes **ZwGetNotificationResourceManager** to time out five seconds after it is called.

A positive value specifies an absolute time, which is actually relative to 00:00, January 1, 1601. If an absolute time value is specified, the operating system adds the absolute time value to the time value that represents 00:00, January 1, 1601.

If the caller specifies a zero value (instead of a **NULL** pointer), **ZwGetNotificationResourceManager** returns immediately, whether a notification is available or not.

`ReturnLength`

An optional pointer to a variable. If this pointer is not **NULL**, and if the *NotificationLength* parameter's value is too small, **ZwGetNotificationResourceManager** supplies the required length in the variable and returns STATUS_BUFFER_TOO_SMALL.

`Asynchronous`

A ULONG value that must be zero. **ZwGetNotificationResourceManager** does not support asynchronous notifications. Use TmEnableCallbacks to enable asynchronous notifications.

`AsynchronousContext`

A pointer to a ULONG value. This pointer must be **NULL**.

Return Value

ZwGetNotificationResourceManager returns STATUS_SUCCESS if the operation succeeds and a notification is available. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_TIMEOUT	The time-out interval that *Timeout* specifies elapsed before a notification became available.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the resource manager object.
STA TUS _BU FFE R_T OO_ SM ALL	The *NotificationLength* parameter's value is too small.

The routine might return other NTSTATUS values.

Remarks

Use the **ZwGetNotificationResourceManager** routine to obtain notifications synchronously. Use the TmEnableCallbacks routine to enable asynchronous notifications.

The received TRANSACTION_NOTIFICATION structure contains the enlistment key that the resource manager specified when it called ZwCreateEnlistment. You can use the enlistment key to identify the enlistment that the notification applies to.

For more information about the **ZwGetNotificationResourceManager** routine, see Creating a Resource Manager.

NtGetNotificationResourceManager and **ZwGetNotificationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	"= PASSIVE_LEVEL"
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_NOTIFICATION

TmEnableCallbacks

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwCreateResourceManager

ZwOpenResourceManager

ZwLoadDriver function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwLoadDriver** routine loads a driver into the system.

Syntax

```
NTSYSAPI NTSTATUS ZwLoadDriver(
  PUNICODE_STRING DriverServiceName
);
```

Parameters

`DriverServiceName`

Pointer to a counted Unicode string that specifies a path to the driver's registry key, \Registry\Machine\System\CurrentControlSet\Services\<*DriverName*>, where <*DriverName*> is the name of the driver.

Return Value

ZwLoadDriver returns STATUS_SUCCESS or an appropriate error NTSTATUS value.

Remarks

ZwLoadDriver dynamically loads a device or file system driver into the currently running system.

> **Note** If the system is running in safe mode, and the driver fails to load because it is not on the safe mode list, **ZwLoadDriver** returns STATUS_SUCCESS.

A minifilter should use FltLoadFilter instead of **ZwLoadDriver** to load a supporting minifilter.

> **Note** If the call to the **ZwLoadDriver** function occurs in user mode, you should use the name "**NtLoadDriver**" instead of "**ZwLoadDriver**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows XP and later versions of Windows.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FltLoadFilter

RtlInitUnicodeString

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwUnloadDriver

ZwMakeTemporaryObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwMakeTemporaryObject** routine changes the attributes of an object to make it temporary.

Syntax

```
NTSYSAPI NTSTATUS ZwMakeTemporaryObject(
  HANDLE Handle
);
```

Parameters

`Handle`

Handle to an object of any type.

Return Value

ZwMakeTemporaryObject returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure.

Remarks

ZwMakeTemporaryObject is a generic routine that operates on any type of object.

An object is permanent if it was created with the OBJ_PERMANENT object attribute specified. (For more information about object attributes, see InitializeObjectAttributes.) A permanent object is created with a reference count of 1, so it is not deleted when a driver dereferences it.

An object is temporary if it is not permanent. **ZwMakeTemporaryObject** turns the specified object into a temporary object. If the object is already temporary, this routine does nothing.

A temporary object has a name only as long as its handle count is greater than zero. When the handle count reaches zero, the system deletes the object name and appropriately adjusts the object's pointer count.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtMakeTemporaryObject**" instead of "**ZwMakeTemporaryObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateDirectoryObject

ZwCreateFile

ZwMapViewOfSection function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwMapViewOfSection** routine maps a view of a section into the virtual address space of a subject process.

Syntax

```
NTSYSAPI NTSTATUS ZwMapViewOfSection(
  HANDLE          SectionHandle,
  HANDLE          ProcessHandle,
  PVOID           *BaseAddress,
  ULONG_PTR       ZeroBits,
  SIZE_T          CommitSize,
  PLARGE_INTEGER  SectionOffset,
  PSIZE_T         ViewSize,
  SECTION_INHERIT InheritDisposition,
  ULONG           AllocationType,
  ULONG           Win32Protect
);
```

Parameters

`SectionHandle`

Handle to a section object. This handle is created by a successful call to ZwCreateSection or ZwOpenSection.

`ProcessHandle`

Handle to the object that represents the process that the view should be mapped into. Use the ZwCurrentProcess macro to specify the current process. The handle must have been opened with PROCESS_VM_OPERATION access (described in the Microsoft Windows SDK documentation).

`BaseAddress`

Pointer to a variable that receives the base address of the view. If the value of this parameter is not **NULL**, the view is allocated starting at the specified virtual address rounded down to the next 64-kilobyte address boundary.

`ZeroBits`

Specifies the number of high-order address bits that must be zero in the base address of the section view. The value of this parameter must be less than 21 and is used only if *BaseAddress* is **NULL**—in other words, when the caller allows the system to determine where to allocate the view.

`CommitSize`

Specifies the size, in bytes, of the initially committed region of the view. *CommitSize* is meaningful only for page-file backed sections and is rounded up to the nearest multiple of PAGE_SIZE. (For sections that map files, both the data and the image are committed at section-creation time.)

`SectionOffset`

A pointer to a variable that receives the offset, in bytes, from the beginning of the section to the view. If this pointer is not **NULL**, the offset is rounded down to the next allocation-granularity size boundary.

`ViewSize`

A pointer to a SIZE_T variable. If the initial value of this variable is zero, **ZwMapViewOfSection** maps a view of the section that starts at *SectionOffset* and continues to the end of the section. Otherwise, the initial value specifies the view's size, in bytes. **ZwMapViewOfSection** always rounds this value up to the nearest multiple of PAGE_SIZE before mapping the view.

On return, the value receives the actual size, in bytes, of the view.

`InheritDisposition`

Specifies how the view is to be shared with child processes. The possible values are:

ViewShare

The view will be mapped into any child processes that are created in the future.

ViewUnmap

The view will not be mapped into child processes.

Drivers should typically specify **ViewUnmap** for this parameter.

`AllocationType`

Specifies a set of flags that describes the type of allocation to be performed for the specified region of pages. The valid flags are MEM_LARGE_PAGES, MEM_RESERVE, and MEM_TOP_DOWN. Although MEM_COMMIT is not allowed, it is implied unless MEM_RESERVE is specified. For more information about the MEM_*XXX* flags, see the description of the VirtualAlloc routine.

`Win32Protect`

Specifies the type of protection for the region of initially committed pages. Device and intermediate drivers should set this value to PAGE_READWRITE.

Return Value

ZwMapViewOfSection returns an NTSTATUS value. Possible return values include the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully performed the requested operation.
STATUS_CONFLICTING_ADDRESSES	The specified address range conflicts with an address range already reserved, or the specified cache attribute type conflicts with the address range's existing cache attribute. For example, if the memory being mapped lies within a large page that is already mapped as fully cached, then it is illegal to request to map this memory as noncached or write combined.

STA TUS _IN VAL ID_ PA GE_ PR OTE CTI ON	The value specified for the *Protect* parameter is invalid.
STA TUS _SE CTI ON_ PR OTE CTI ON	The value specified for the *AllocationType* parameter is incompatible with the protection type specified when the section was created.

Remarks

Several different views of a section can be concurrently mapped into the virtual address space of one or more processes.

If the specified section does not exist or the access requested is not allowed, **ZwMapViewOfSection** returns an error.

Do not use **ZwMapViewOfSection** to map a memory range from **\Device\PhysicalMemory** into user mode—unless your driver has directly allocated the memory range through MmAllocatePagesForMdl or another method guaranteeing that no other system component has mapped the same memory range with a different MEMORY_CACHING_TYPE value.

User applications cannot access **\Device\PhysicalMemory** directly starting with Windows Server 2003 with Service Pack 1 (SP1) and can access it only if the driver passes a handle to the application.

For more information about section objects, see Section Objects and Views.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtMapViewOfSection**" instead of "**ZwMapViewOfSection**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

MEMORY_CACHING_TYPE

MmAllocatePagesForMdl

Using Nt and Zw Versions of the Native System Services Routines

VirtualAlloc

ZwCurrentProcess

ZwOpenSection

ZwUnmapViewOfSection

ZwOpenEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenEnlistment** routine obtains a handle to an existing enlistment object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwOpenEnlistment(
  PHANDLE            EnlistmentHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             RmHandle,
  LPGUID             EnlistmentGuid,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`EnlistmentHandle`

A pointer to a caller-allocated variable that receives a handle to an enlistment object if the call to **ZwOpenEnlistment** succeeds.

`DesiredAccess`

An ACCESS_MASK-typed value that specifies the requested access to the enlistment object. For more information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateEnlistment. This parameter cannot be zero.

`RmHandle`

`EnlistmentGuid`

A pointer to a GUID that identifies the enlistment. For more information, see the following Remarks section.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. This parameter is optional and can be **NULL**.

Return Value

ZwOpenEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.
STATUS_INVALID_PARAMETER	The *DesiredAccess* parameter's value is zero, or the *EnlistmentGuid* parameter's value is **NULL**.
STATUS_ENLISTMENT_NOT_FOUND	The enlistment that the *EnlistmentGuid* parameter specifies does not exist for the resource manager that the *RmHandle* parameter specifies.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

Typically, a TPS component calls **ZwOpenEnlistment** after it receives an enlistment GUID from another TPS component that had previously called ZwCreateEnlistment. Most TPS designs do not require calling **ZwOpenEnlistment**.

A resource manager that calls **ZwOpenEnlistment** must eventually call ZwClose to close the object handle.

For more information about **ZwOpenEnlistment**, see Enlistment Objects.

NtOpenEnlistment and **ZwOpenEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateEnlistment

ZwCreateResourceManager

ZwOpenResourceManager

ZwQueryInformationEnlistment

ZwSetInformationEnlistment

ZwOpenEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenEvent** routine opens a handle to an existing named event object with the specified desired access.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwOpenEvent(
  PHANDLE            EventHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`EventHandle`

A pointer to a variable that will receive the event object handle. The handle includes bookkeeping information, such as a reference count and security context.

`DesiredAccess`

The ACCESS_MASK value that represents the desired types of access for the event object. The following table contains the event-specific ACCESS_MASK values.

VALUE	DESIRED ACCESS
EVENT_QUERY_STATE	Query the state of the event object.
EVENT_MODIFY_STATE	Modify the state of the event object.
EVENT_ALL_ACCESS	All possible access rights to the event object.

`ObjectAttributes`

A pointer to the object attributes structure that the caller supplied to be used for the specified object. These attributes would include the **ObjectName** and the handle attributes, for example. This parameter is initialized by calling the InitializeObjectAttributes macro.

Return Value

ZwOpenEvent returns STATUS_SUCCESS or an appropriate error status. This routine might return one of the following error status codes:

RETURN CODE	DESCRIPTION

STATUS_INSUFFICIENT_RESOURCES	Resources that this function requires could not be allocated.
STATUS_INVALID_PARAMETER	The *ObjectAttributes* structure was not supplied or contained an invalid parameter value.
STATUS_OBJECT_NAME_INVALID	The *ObjectAttributes* parameter contained an **ObjectName** in the OBJECT_ATTRIBUTES structure that was invalid.
STATUS_OBJECT_PATH_SYNTAX_BAD	The object path contained incorrect syntax. For example, the *ObjectAttributes* parameter does not contain a **RootDirectory** member, but the **ObjectName** member in the **OBJECT_ATTRIBUTES** structure is an empty string or does not start with an OBJECT_NAME_PATH_SEPARATOR character.
STATUS_OBJECT_PATH_NOT_FOUND	The path to the object was not found, indicating that it does not exist.

STA TUS _OB JEC T_N AM E_N OT_ FOU ND	The object name was not found, indicating that the object does not exist.
STA TUS _PRI VILE GE_ NO T_H ELD	The caller did not have the required privilege to create a handle with the access specified in the *DesiredAccess* parameter.

Remarks

ZwOpenEvent opens an existing named event object and creates a handle to the object with the specified desired access.

ZwOpenEvent can open either notification or synchronization events.

Events are used to coordinate execution. File system drivers can use events to enable a caller to wait for completion of the requested operation until the given event is set to the Signaled state.

Notification events can be used to notify one or more threads of execution that an event has occurred. Synchonization events can be used in the serialization of access to hardware between two otherwise unrelated drivers.

> **Note** If the call to the **ZwOpenEvent** function occurs in user mode, you should use the name "**NtOpenEvent**" instead of "**ZwOpenEvent**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows Server 2003.
Target Platform	Universal
Header	wdm.h (include Wdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

IoCreateNotificationEvent

IoCreateSynchronizationEvent

KeClearEvent

KeResetEvent

KeSetEvent

KeWaitForSingleObject

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateEvent

ZwSetEvent

ZwWaitForSingleObject

ZwOpenFile function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenFile** routine opens an existing file, directory, device, or volume.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenFile(
  PHANDLE            FileHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PIO_STATUS_BLOCK   IoStatusBlock,
  ULONG              ShareAccess,
  ULONG              OpenOptions
);
```

Parameters

`FileHandle`

Pointer to a HANDLE variable that receives a handle to the file.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. For more information, see the *DesiredAccess* parameter of ZwCreateFile.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested operation.

`ShareAccess`

Specifies the type of share access for the file. For more information, see the *ShareAccess* parameter of ZwCreateFile.

`OpenOptions`

Specifies the options to apply when opening the file. For more information, see the *CreateOptions* parameter of ZwCreateFile.

Return Value

ZwOpenFile returns STATUS_SUCCESS or the appropriate NTSTATUS error code. In the latter case, the caller can find more information about the cause of the failure by checking the *IoStatusBlock* parameter.

Remarks

ZwOpenFile supplies a handle that the caller can use to manipulate a file's data, or the file object's state and attributes. **ZwOpenFile** provides a subset of the functionality provided by ZwCreateFile. For more information, see Using Files in a Driver.

Once the handle pointed to by *FileHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

Callers of **ZwOpenFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenFile**" instead of "**ZwOpenFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateFile

ZwOpenKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenKey** routine opens an existing registry key.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenKey(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`KeyHandle`

Pointer to the HANDLE variable that receives the handle to the key.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. For more information, see the *DesiredAccess* parameter of ZwCreateKey.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

Return Value

ZwOpenKey returns STATUS_SUCCESS if the given key was opened. Otherwise, it can return an error status, including the following:

STATUS_INVALID_HANDLE

STATUS_ACCESS_DENIED

Remarks

ZwOpenKey supplies a handle that the caller can use to manipulate a registry key. The routine provides a subset of the functionality of ZwCreateKey. For more information, see Using the Registry in a Driver.

If the specified key does not exist, **ZwOpenKey** returns an error status and does not return a key handle.

Once the handle pointed to by *KeyHandle* is no longer in use, the driver must call ZwClose to close it.

ZwOpenKey ignores the security information in the structure that the *ObjectAttributes* parameter points to.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about working with registry keys, see Using the Registry in a Driver.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenKey**" instead of "**ZwOpenKey**".

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate, ZwRegistryOpen(storport)

See Also

ACCESS_MASK

InitializeObjectAttributes

ZwCreateKey

ZwDeleteKey

ZwEnumerateKey

ZwEnumerateValueKey

ZwFlushKey

ZwQueryKey

ZwQueryValueKey

ZwSetValueKey

ZwOpenKeyEx function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenKeyEx** routine opens an existing registry key.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenKeyEx(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              OpenOptions
);
```

Parameters

`KeyHandle`

A pointer to a HANDLE variable into which the routine writes the handle to the key.

`DesiredAccess`

Specifies the type of access to the key that the caller requests. This parameter is an ACCESS_MASK value. For more information, see the description of the *DesiredAccess* parameter of the ZwCreateKey routine.

`ObjectAttributes`

A pointer to the object attributes of the key being opened. This parameter points to an OBJECT_ATTRIBUTES structure that must have been previously initialized by the InitializeObjectAttributes routine. The caller must specify the name of the registry key as the *ObjectName* parameter in the call to **InitializeObjectAttributes**. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`OpenOptions`

Specifies the options to apply when opening the key. Set this parameter to zero or to the bitwise OR of one or more of the following REG_OPTION_*XXX* flag bits:

***OPENOPTIONS* FLAG**	**DESCRIPTION**
REG_OPTION_OPEN_LINK	The key is a symbolic link. This flag is not used by device and intermediate drivers.
REG_OPTION_BACKUP_RESTORE	The key should be opened with special privileges that allow backup and restore operations. This flag is not used by device and intermediate drivers.

Return Value

ZwOpenKeyEx returns STATUS_SUCCESS if the call succeeds in opening the key. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER_4	The *OpenOptions* parameter specifies invalid options.
STATUS_OBJECT_PATH_SYNTAX_BAD	The registry path in the object attributes is invalid.
STATUS_OBJECT_NAME_NOT_FOUND	The registry key name in the object attributes was not found.
STATUS_OBJECT_TYPE_MISMATCH	The named registry key is a symbolic link, but the REG_OPTION_OPEN_LINK flag bit is not set in *OpenOptions*.
STATUS_ACCESS_DENIED	The caller did not have the required access rights to open a handle for the named registry key.

STA **TUS** **_IN** **SUF** **FICI** **ENT** **_RE** **SO** **URC** **ES**	A memory allocation operation failed.

Remarks

This routine supplies a handle with which the caller can access a registry key. If the specified key does not exist, the routine returns an error status value and does not supply a key handle.

The ZwOpenKey routine is similar to **ZwOpenKeyEx** but does not accept an *OpenOptions* parameter. The *OpenOptions* parameter of **ZwOpenKeyEx** enables the caller to open a key that is a symbolic link, or to open a key for backup and restore operations. A call to **ZwOpenKeyEx** with *OpenOptions* = 0 is equivalent to a call to **ZwOpenKey**.

After the handle pointed to by *KeyHandle* is no longer in use, the driver must call ZwClose to close it.

ZwOpenKeyEx ignores the security information in the structure that the *ObjectAttributes* parameter points to.

If the kernel-mode caller is not running in a system thread context, it must ensure that any handles it creates are kernel handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about working with registry keys in kernel mode, see Using the Registry in a Driver.

Requirements

Minimum supported client	Available in Windows 7 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

ZwClose

ZwOpenKey

ZwOpenKeyTransacted function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwOpenKeyTransacted** routine opens an existing registry key and associates the key with a transaction.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenKeyTransacted(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  HANDLE             TransactionHandle
);
```

Parameters

`KeyHandle`

A pointer to a HANDLE variable into which the routine writes the handle to the key.

`DesiredAccess`

Specifies the type of access to the key that the caller requests. This parameter is an ACCESS_MASK value. For more information, see the description of the *DesiredAccess* parameter of the ZwCreateKey routine.

`ObjectAttributes`

A pointer to the object attributes of the key being opened. This parameter points to an OBJECT_ATTRIBUTES structure that must have been previously initialized by the InitializeObjectAttributes routine. The caller must specify the name of the registry key as the *ObjectName* parameter in the call to **InitializeObjectAttributes**. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`TransactionHandle`

A handle to a transaction object. To obtain this handle, you can call the ZwCreateTransaction routine. Or, if you have a pointer to a transaction object, you can supply the pointer to the ObOpenObjectByPointer routine to obtain the corresponding transaction handle.

Return Value

ZwOpenKeyTransacted returns STATUS_SUCCESS if the call successfully opens the key. Possible error return values include the following:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The *ObjectAttributes* parameter is **NULL** or points to invalid information.
STATUS_OBJECT_PATH_SYNTAX_BAD	The registry path in the object attributes is invalid.
STATUS_OBJECT_NAME_NOT_FOUND	The registry path in the object attributes was not found.
STATUS_ACCESS_DENIED	The caller did not have the required access rights to open a handle for the named registry key.
STATUS_INSUFFICIENT_RESOURCES	A memory allocation operation failed.

Remarks

This routine provides a handle with which the caller can access a registry key. Additionally, this routine associates the key with an active transaction.

After the handle that is pointed to by *KeyHandle* is no longer being used, the driver must call the ZwClose routine to close it.

If the specified key does not exist in the registry, **ZwOpenKeyTransacted** returns an error status value and does not supply a key handle. Unlike the ZwCreateKeyTransacted routine, the **ZwOpenKeyTransacted** routine does not create the specified key if the key does not exist. Both **ZwCreateKeyTransacted** and **ZwOpenKeyTransacted** associate a registry key with a transaction.

The ZwOpenKey routine is similar to **ZwOpenKeyTransacted**, but does not associate a key with a transaction.

After a kernel-mode driver obtains a handle to a transaction (for example, by calling **ZwCreateTransaction**), the driver can perform a series of registry operations that are part of this transaction. The driver can close the transaction either by committing to the changes that were made in the transaction or by rolling back the transaction.

After the driver successfully completes all registry operations that are part of a transaction, it can call the ZwCommitTransaction routine to commit to the changes. The driver can call the ZwRollbackTransaction routine to roll back the transaction.

During a transaction, a registry operation is part of the transaction if the system call that performs the operation meets either of the following conditions:

- The call specifies, as an input parameter, the transaction handle. For example, calls to **ZwCreateKeyTransacted** and **ZwOpenKeyTransacted** can associate one or more keys with the transaction.
- The call specifies, as an input parameter, a registry key handle that was obtained by a call to **ZwCreateKeyTransacted** or **ZwOpenKeyTransacted** to which the transaction handle was supplied. For example, a call to the ZwSetValueKey routine can use a key handle that was obtained in this way to set the value of a registry key as part of a transaction.

For more information about kernel-mode transactions, see Using Kernel Transaction Manager.

ZwOpenKeyTransacted ignores the security information in the structure that the *ObjectAttributes* parameter points to.

If the kernel-mode caller is not running in a system thread context, it must ensure that any handles it creates are kernel handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about how to work with registry keys in kernel mode, see Using the Registry in a Driver.

Requirements

Minimum supported client	Available in Windows Vista and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

ObOpenObjectByPointer

ZwClose

ZwCommitTransaction

ZwCreateKey

ZwCreateKeyTransacted

ZwCreateTransaction

ZwOpenKey

ZwOpenKeyTransactedEx

ZwRollbackTransaction

ZwSetValueKey

ZwOpenKeyTransactedEx function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwOpenKeyTransactedEx** routine opens an existing registry key and associates the key with a transaction.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenKeyTransactedEx(
  PHANDLE            KeyHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  ULONG              OpenOptions,
  HANDLE             TransactionHandle
);
```

Parameters

`KeyHandle`

A pointer to a HANDLE variable into which the routine writes the handle to the key.

`DesiredAccess`

Specifies the type of access to the key that the caller requests. This parameter is an ACCESS_MASK value. For more information, see the description of the *DesiredAccess* parameter of the ZwCreateKey routine.

`ObjectAttributes`

A pointer to the object attributes of the key being opened. This parameter points to an OBJECT_ATTRIBUTES structure that must have been previously initialized by the InitializeObjectAttributes routine. The caller must specify the name of the registry key as the *ObjectName* parameter in the call to **InitializeObjectAttributes**. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

`OpenOptions`

Specifies the options to apply when opening the key. Set this parameter to zero or to the bitwise OR of one or more of the following REG_OPTION_*XXX* flag bits.

***OPENOPTIONS* FLAG**	**DESCRIPTION**
REG_OPTION_OPEN_LINK	The key is a symbolic link. This flag is not used by device and intermediate drivers.
REG_OPTION_BACKUP_RESTORE	The key should be opened with special privileges that allow backup and restore operations. This flag is not used by device and intermediate drivers.

`TransactionHandle`

A handle to a transaction object. To obtain this handle, you can call the ZwCreateTransaction routine. Or, if you have a pointer to a transaction object, you can supply the pointer to the ObOpenObjectByPointer routine to obtain the corresponding transaction handle.

Return Value

ZwOpenKeyTransactedEx returns STATUS_SUCCESS if the call successfully opens the key. Possible error return values include the following:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The *ObjectAttributes* parameter is **NULL** or points to invalid information.
STATUS_INVALID_PARAMETER_4	The *OpenOptions* parameter value specifies invalid options.
STATUS_OBJECT_PATH_SYNTAX_BAD	The registry path in the object attributes is invalid.
STATUS_OBJECT_NAME_NOT_FOUND	The registry path in the object attributes was not found.

STA TUS _AC CES S_D ENI ED	The caller did not have the required access rights to open a handle for the named registry key.
STA TUS _IN SUF FICI ENT _RE SO URC ES	A memory allocation operation failed.

Remarks

This routine provides a handle with which the caller can access a registry key. Additionally, this routine associates the key with an active transaction.

After the handle that is pointed to by *KeyHandle* is no longer being used, the driver must call the ZwClose routine to close it.

If the specified key does not exist in the registry, **ZwOpenKeyTransactedEx** returns an error status value and does not supply a key handle. Unlike the ZwCreateKeyTransacted routine, the **ZwOpenKeyTransactedEx** routine does not create the specified key if the key does not exist. Both **ZwCreateKeyTransacted** and **ZwOpenKeyTransactedEx** associate a registry key with a transaction.

The ZwOpenKeyEx routine is similar to **ZwOpenKeyTransactedEx**, but does not associate a key with a transaction.

The ZwOpenKeyTransacted routine is similar to **ZwOpenKeyTransactedEx** but does not accept an *OpenOptions* parameter. The *OpenOptions* parameter of **ZwOpenKeyTransactedEx** enables the caller to open a key that is a symbolic link, or to open a key for backup and restore operations. A call to **ZwOpenKeyTransactedEx** with *OpenOptions* parameter set to zero is equivalent to a call to **ZwOpenKeyTransacted**.

After a kernel-mode driver obtains a handle to a transaction (for example, by calling **ZwCreateTransaction**), the driver can perform a series of registry operations that are part of this transaction. The driver can close the transaction either by committing to the changes that were made in the transaction or by rolling back the transaction.

After the driver successfully completes all registry operations that are part of a transaction, it can call the ZwCommitTransaction routine to commit to the changes. The driver can call the ZwRollbackTransaction routine to roll back the transaction.

During a transaction, a registry operation is part of the transaction if the system call that performs the operation meets either of the following conditions:

- The call specifies, as an input parameter, the transaction handle. For example, calls to **ZwCreateKeyTransacted** and **ZwOpenKeyTransactedEx** can associate one or more keys with the transaction.
- The call specifies, as an input parameter, a registry key handle that was obtained by a call to

ZwCreateKeyTransacted or **ZwOpenKeyTransactedEx** to which the transaction handle was supplied. For example, a call to the ZwSetValueKey routine can use a key handle that was obtained in this way to set the value of a registry key as part of a transaction.

For more information about kernel-mode transactions, see Using Kernel Transaction Manager.

ZwOpenKeyTransactedEx ignores the security information in the structure that the *ObjectAttributes* parameter points to.

If the kernel-mode caller is not running in a system thread context, it must ensure that any handles it creates are kernel handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

For more information about how to work with registry keys in kernel mode, see Using the Registry in a Driver.

Requirements

Minimum supported client	Available in Windows 7 and later versions of the Windows operating system.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

ObOpenObjectByPointer

ZwClose

ZwCommitTransaction

ZwCreateKey

ZwCreateKeyTransacted

ZwCreateTransaction

ZwOpenKeyEx

ZwOpenKeyTransacted

ZwRollbackTransaction

ZwSetValueKey

ZwOpenResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenResourceManager** routine returns a handle to an existing resource manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwOpenResourceManager(
  PHANDLE            ResourceManagerHandle,
  ACCESS_MASK        DesiredAccess,
  HANDLE             TmHandle,
  LPGUID             ResourceManagerGuid,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`ResourceManagerHandle`

A pointer to a caller-allocated variable that receives the resource manager handle if the call to **ZwOpenResourceManager** succeeds.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the resource manager object. For more information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateResourceManager. This parameter cannot be zero.

`TmHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager.

`ResourceManagerGuid`

A pointer to the GUID that identifies the resource manager to open.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. This parameter is optional and can be **NULL**.

Return Value

ZwOpenResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The handle that *TmHandle* specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	The handle that *TmHandle* specifies is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the specified transaction manager object.
STATUS_INVALID_PARAMETER	The *DesiredAccess* parameter is zero or the *ResourceManagerGuid* parameter is invalid.
STATUS_RESOURCEMANAGER_NOT_FOUND	The specified resource manager could not be found.

STA TUS _TR AN SAC TIO NM AN AGE R_N OT_ ON LIN E	The specified transaction manager is not online.

The routine might return other NTSTATUS values.

Remarks

Typically, a TPS component calls **ZwOpenResourceManager** after it receives an enlistment GUID from another TPS component that had previously called ZwCreateResourceManager. Most TPS designs do not require calling **ZwOpenResourceManager**.

A resource manager that calls **ZwOpenResourceManager** must eventually call ZwClose to close the object handle.

For more information about **ZwOpenResourceManager**, see KTM Objects.

NtOpenResourceManager and **ZwOpenResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateResourceManager

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwOpenSection function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenSection** routine opens a handle for an existing section object.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenSection(
  PHANDLE            SectionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`SectionHandle`

Pointer to a HANDLE variable that receives a handle to the section object.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. For more information, see the *DesiredAccess* parameter of ZwCreateSection.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

Return Value

ZwOpenSection returns STATUS_SUCCESS on success, or the appropriate error code on failure. Possible return values include:

Remarks

If the section does not exist or the system did not grant the requested access, the operation fails.

Once the handle pointed to by *SectionHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtOpenSection**" instead of "**ZwOpenSection**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateSection

ZwMapViewOfSection

ZwUnmapViewOfSection

ZwOpenSymbolicLinkObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenSymbolicLinkObject** routine opens an existing symbolic link.

Syntax

```
NTSYSAPI NTSTATUS ZwOpenSymbolicLinkObject(
  PHANDLE            LinkHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes
);
```

Parameters

`LinkHandle`

Pointer to a HANDLE variable that receives a handle to the symbolic link object.

`DesiredAccess`

Specifies an ACCESS_MASK value that determines the requested access to the object. The caller usually specified GENERIC_READ, so that the handle can be passed to ZwQuerySymbolicLinkObject.

`ObjectAttributes`

Pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use InitializeObjectAttributes to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**.

Return Value

ZwOpenSymbolicLinkObject returns STATUS_SUCCESS on success or the appropriate error status.

Remarks

Once the handle pointed to by *LinkHandle* is no longer in use, the driver must call ZwClose to close it.

If the caller is not running in a system thread context, it must ensure that any handles it creates are private handles. Otherwise, the handle can be accessed by the process in whose context the driver is running. For more information, see Object Handles.

> **Note** If the call to the **ZwOpenSymbolicLinkObject** function occurs in user mode, you should use the name "NtOpenSymbolicLinkObject" instead of "**ZwOpenSymbolicLinkObject**".

NtOpenSymbolicLinkObject and **ZwOpenSymbolicLinkObject** are two versions of the same Windows Native System Services routine. The **NtOpenSymbolicLinkObject** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwOpenSymbolicLinkObject** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine

can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

Using Nt and Zw Versions of the Native System Services Routines

ZwQuerySymbolicLinkObject

ZwOpenTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenTransaction** routine obtains a handle to an existing transaction object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwOpenTransaction(
  PHANDLE            TransactionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  LPGUID             Uow,
  HANDLE             TmHandle
);
```

Parameters

`TransactionHandle`

A pointer to a caller-allocated variable that receives a handle to the transaction object if **ZwOpenTransaction** returns STATUS_SUCCESS.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction object. For information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateTransaction.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object's attributes. Use the InitializeObjectAttributes routine to initialize this structure, but specify only that routine's *InitializedAttributes* and *Attributes* parameters. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE flag in the *Attributes* parameter. The *ObjectAttributes* parameter is optional and can be **NULL**.

`Uow`

A pointer to a GUID that is a transaction object's unit of work (UOW) identifier. This GUID identifies the transaction object to open.

`TmHandle`

A handle to a transaction manager object. If this parameter is not **NULL**, KTM searches only for transaction objects that belong to the specified transaction manager object. If this parameter is **NULL**, KTM searches all transaction objects.

Return Value

ZwCreateTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_INVALID_PARAMETER	The *DesiredAccess* or *Uow* parameter was zero.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_TRANSACTION_NOT_FOUND	KTM could not find the transaction object.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

Typically, a resource manager calls **ZwOpenTransaction** after it receives a transaction UOW from a transactional

client that had previously called ZwCreateTransaction.

For more information about **ZwOpenTransaction**, see Creating a Resource Manager.

NtOpenTransaction and **ZwOpenTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ACCESS_MASK

InitializeObjectAttributes

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwCommitTransaction

ZwCreateTransaction

ZwQueryInformationTransaction

ZwRollbackTransaction

ZwOpenTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwOpenTransactionManager** routine obtains a handle to an existing transaction manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwOpenTransactionManager(
  PHANDLE            TmHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PUNICODE_STRING    LogFileName,
  LPGUID             TmIdentity,
  ULONG              OpenOptions
);
```

Parameters

`TmHandle`

A pointer to a caller-allocated variable that receives a handle to the transaction manager object if **ZwOpenTransactionManager** returns STATUS_SUCCESS.

`DesiredAccess`

An ACCESS_MASK value that specifies the caller's requested access to the transaction manager object. For information about how to specify this parameter, see the *DesiredAccess* parameter of ZwCreateTransactionManager.

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that specifies the object name and other attributes. Use the InitializeObjectAttributes routine to initialize this structure. If the caller is not running in a system thread context, it must set the OBJ_KERNEL_HANDLE attribute when it calls **InitializeObjectAttributes**. This parameter is optional and can be **NULL**.

`LogFileName`

A pointer to a UNICODE_STRING structure that contains the path and file name of the log file stream that was created when the transaction manager object was created. For more information, see the *LogFileName* parameter of **ZwCreateTransactionManager**. This parameter is optional and can be **NULL**.

`TmIdentity`

A pointer to a GUID that identifies the transaction manager object. This parameter is optional and can be **NULL**.

`OpenOptions`

This parameter is not used and must be zero.

Return Value

ZwOpenTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The value of an input parameter is invalid.
STATUS_INSUFFICIENT_RESOURCES	KTM could not allocate system resources (typically memory).
STATUS_OBJECT_NAME_INVALID	The object name that the *ObjectAttributes* parameter specifies is invalid.
STATUS_LOG_CORRUPTION_DETECTED	KTM encountered an error while creating or opening the log file.
STATUS_ACCESS_DENIED	The value of the *DesiredAccess* parameter is invalid.

The routine might return other NTSTATUS values.

Remarks

The caller can identify which transaction manager object to open by using one of the following three techniques:

- Use the *LogFileName* parameter to specify the path and file name of a log file stream that was created when the transaction manager object was created.
- Use the *TmIdentity* parameter to specify the GUID that identifies the transaction manager object.
- Use the *ObjectAttributes* parameter to supply an OBJECT_ATTRIBUTES structure that contains the object name that the caller previously specified to the ZwCreateTransactionManager routine.

You must specify only one of the above-listed parameters (an object name, a log file name, or a GUID) and set the other two parameters to **NULL**.

Your TPS component must call ZwRecoverTransactionManager after it has called **ZwOpenTransactionManager**.

A TPS component that calls **ZwOpenTransactionManager** must eventually call ZwClose to close the object handle.

For more information about how to use **ZwOpenTransactionManager**, see Transaction Manager Objects and Creating a Resource Manager.

NtOpenTransactionManager and **ZwOpenTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

InitializeObjectAttributes

OBJECT_ATTRIBUTES

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateTransactionManager

ZwQueryInformationTransactionManager

ZwPrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrepareComplete** routine notifies KTM that the calling resource manager has finished preparing a transaction's data.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwPrepareComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwPrepareComplete** after it has finished servicing a TRANSACTION_NOTIFY_PREPARE notification.

After a resource manager has called **ZwPrepareComplete**, it cannot roll back the transaction.

For more information about **ZwPrepareComplete**, see Handling Commit Operations.

NtPrepareComplete and **ZwPrepareComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrepareComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrePrepareComplete

ZwPrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrepareEnlistment** routine initiates the prepare operation for a specified enlistment's transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwPrepareEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_PREPARE_COMPLETE notifications.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment's transaction is not in a state that allows it to enter the prepare phase.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwPrepareEnlistment**.

The **ZwPrepareEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_PREPARE notifications to all resource managers that have enlisted in the transaction.

Callers of **ZwPrepareEnlistment** must register to receive TRANSACTION_NOTIFY_PREPARE_COMPLETE notifications.

For more information about **ZwPrepareEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

NtPrePrepareEnlistment and **ZwPrePrepareEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrepareEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrePrepareComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrePrepareComplete** routine notifies KTM that the calling resource manager has finished preliminary preparation of a transaction's data.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwPrePrepareComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrePrepareComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwPrePrepareComplete** after it has finished servicing a TRANSACTION_NOTIFY_PREPREPARE notification.

For more information about **ZwPrePrepareComplete**, see Handling Commit Operations.

NtPrePrepareComplete and **ZwPrePrepareComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrePrepareComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

ZwPrePrepareEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwPrePrepareEnlistment** routine initiates the pre-prepare operation for a specified enlistment's transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwPrePrepareEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The object must represent a superior enlistment and the handle must have ENLISTMENT_SUPERIOR_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwPrePrepareEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_ENLISTMENT_NOT_SUPERIOR	The caller is not a superior transaction manager for the enlistment.

STATUS_TRANSACTION_RESPONSE_NOT_ENLISTED	The caller did not register to receive TRANSACTION_NOTIFY_PREPREPARE_COMPLETE notifications.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment's transaction is not in a state that allows it to enter the prepare phase.

The routine might return other NTSTATUS values.

Remarks

Only superior transaction managers can call **ZwPrePrepareEnlistment**.

Callers of **ZwPrePrepareEnlistment** must register to receive TRANSACTION_NOTIFY_PREPREPARE_COMPLETE notifications.

The **ZwPrePrepareEnlistment** routine causes KTM to send TRANSACTION_NOTIFY_PREPREPARE notifications to all resource managers that have enlisted in the transaction.

For more information about **ZwPrePrepareEnlistment**, see Creating a Superior Transaction Manager and Handling Commit Operations.

NtPrepareEnlistment and **ZwPrepareEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmPrePrepareEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwQueryFullAttributesFile function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryFullAttributesFile** routine supplies network open information for the specified file.

Syntax

```
NTSYSAPI NTSTATUS ZwQueryFullAttributesFile(
  POBJECT_ATTRIBUTES            ObjectAttributes,
  PFILE_NETWORK_OPEN_INFORMATION FileInformation
);
```

Parameters

`ObjectAttributes`

A pointer to an OBJECT_ATTRIBUTES structure that supplies the attributes to be used for the file object.

`FileInformation`

A pointer to a FILE_NETWORK_OPEN_INFORMATION structure that receives the returned file attributes information.

Return Value

ZwQueryFullAttributesFile returns STATUS_SUCCESS on success, or the appropriate error status.

Remarks

Callers of **ZwQueryFullAttributesFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

NtQueryFullAttributesFile and **ZwQueryFullAttributesFile** are two versions of the same Windows Native System Services routine. The **NtQueryFullAttributesFile** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwQueryFullAttributesFile** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows XP.
Target Platform	Universal

Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_NETWORK_OPEN_INFORMATION

OBJECT_ATTRIBUTES

Using Nt and Zw Versions of the Native System Services Routines

ZwQueryInformationEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationEnlistment** routine retrieves information about a specified enlistment object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwQueryInformationEnlistment(
  HANDLE                      EnlistmentHandle,
  ENLISTMENT_INFORMATION_CLASS EnlistmentInformationClass,
  PVOID                       EnlistmentInformation,
  ULONG                       EnlistmentInformationLength,
  PULONG                      ReturnLength
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_QUERY_INFORMATION access to the object.

`EnlistmentInformationClass`

An ENLISTMENT_INFORMATION_CLASS-typed enumeration value that specifies the information to be obtained. This value must be one of the following values:

- **EnlistmentBasicInformation**
- **EnlistmentRecoveryInformation**

The enumeration's **EnlistmentFullInformation** value is not used with **ZwQueryInformationEnlistment**.

`EnlistmentInformation`

A pointer to a caller-allocated buffer that receives the information that the *EnlistmentInformationClass*parameter specifies. If the *EnlistmentInformationClass* parameter's value is **EnlistmentBasicInformation**, this buffer's structure type must be ENLISTMENT_BASIC_INFORMATION. If the *EnlistmentInformationClass* parameter's value is **EnlistmentRecoveryInformation**, this buffer's type must match the caller-defined type that the caller used when it called ZwSetInformationEnlistment.

`EnlistmentInformationLength`

The length, in bytes, of the buffer that the *EnlistmentInformation* parameter points to.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *EnlistmentInformation*buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *EnlistmentInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The *EnlistmentInformationLength* parameter's value is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call ZwSetInformationEnlistment to set enlistment-specific recovery information for an enlistment object and then call **ZwQueryInformationEnlistment** to retrieve the recovery information.

For more information about **ZwQueryInformationEnlistment**, see Handling Recovery Operations.

Callers of **ZwQueryInformationEnlistment** must be running at IRQL = PASSIVE_LEVEL.

NtQueryInformationEnlistment and **ZwQueryInformationEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ENLISTMENT_BASIC_INFORMATION

ENLISTMENT_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwSetInformationEnlistment

ZwQueryInformationFile function

1/25/2019 • 5 minutes to read • Edit Online

The **ZwQueryInformationFile** routine returns various kinds of information about a file object.

Syntax

```
NTSYSAPI NTSTATUS ZwQueryInformationFile(
  HANDLE                 FileHandle,
  PIO_STATUS_BLOCK       IoStatusBlock,
  PVOID                  FileInformation,
  ULONG                  Length,
  FILE_INFORMATION_CLASS FileInformationClass
);
```

Parameters

`FileHandle`

Handle to a file object. The handle is created by a successful call to ZwCreateFile or ZwOpenFile.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the operation. The **Information** member receives the number of bytes that this routine actually writes to the *FileInformation* buffer.

`FileInformation`

Pointer to a caller-allocated buffer into which the routine writes the requested information about the file object. The *FileInformationClass* parameter specifies the type of information that the caller requests.

`Length`

The size, in bytes, of the buffer pointed to by *FileInformation*.

`FileInformationClass`

Specifies the type of information to be returned about the file, in the buffer that *FileInformation* points to. Device and intermediate drivers can specify any of the following FILE_INFORMATION_CLASS values.

FILE_INFORMATION_CLASS VALUE	TYPE OF INFORMATION RETURNED
FileAccessInformation	A FILE_ACCESS_INFORMATION structure. This structure contains an access mask. For more information about access masks, see ACCESS_MASK.
FileAlignmentInformation	A FILE_ALIGNMENT_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*. This information is useful if the file was opened with the FILE_NO_INTERMEDIATE_BUFFERING flag specified in the *CreateOptions* parameter.

FileAllInformation	A FILE_ALL_INFORMATION structure. By combining several file-information structures into a single structure, **FILE_ALL_INFORMATION** reduces the number of queries required to obtain information about a file.
FileAttributeTagInformation	A FILE_ATTRIBUTE_TAG_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.
FileBasicInformation	A FILE_BASIC_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.
FileEaInformation	A FILE_EA_INFORMATION structure. This structure specifies the size of the extended attributes block that is associated with the file.
FileInternalInformation	A FILE_INTERNAL_INFORMATION structure. This structure specifies a 64-bit file ID that uniquely identifies a file in NTFS. On other file systems, this file ID is not guaranteed to be unique.
FileIoPriorityHintInformation	A FILE_IO_PRIORITY_HINT_INFORMATION structure. The caller must have opened the file with the FILE_READ_DATA flag specified in the *DesiredAccess* parameter.
FileModeInformation	A FILE_MODE_INFORMATION structure. This structure contains a set of flags that specify the mode in which the file can be accessed. These flags are a subset of the options that can be specified in the *CreateOptions* parameter of the IoCreateFile routine.
FileNameInformation	A FILE_NAME_INFORMATION structure. The structure can contain the file's full path or only a portion of it. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*. For more information about the file-name syntax, see the Remarks section later in this topic.
FileNetworkOpenInformation	A FILE_NETWORK_OPEN_INFORMATION structure. The caller must have opened the file with the FILE_READ_ATTRIBUTES flag specified in the *DesiredAccess* parameter.
FilePositionInformation	A FILE_POSITION_INFORMATION structure. The caller must have opened the file with the *DesiredAccess* FILE_READ_DATA or FILE_WRITE_DATA flag specified in the *DesiredAccess* parameter, and with the FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT flag specified in the *CreateOptions*parameter.
FileStandardInformation	A FILE_STANDARD_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*.
FileIsRemoteDeviceInformation	A FILE_IS_REMOTE_DEVICE_INFORMATION structure. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*.

Return Value

ZwQueryInformationFile returns STATUS_SUCCESS or an appropriate NTSTATUS error code.

Remarks

ZwQueryInformationFile returns information about the specified file object. Note that it returns zero in any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by a particular device or file system.

When *FileInformationClass* = **FileNameInformation**, the file name is returned in the FILE_NAME_INFORMATION structure. The precise syntax of the file name depends on a number of factors:

- If you opened the file by submitting a full path to ZwCreateFile, **ZwQueryInformationFile** returns that full path.
- If the **ObjectAttributes->RootDirectory** handle was opened by name in a call to **ZwCreateFile**, and subsequently the file was opened by **ZwCreateFile** relative to this root-directory handle, **ZwQueryInformationFile** returns the full path.
- If the **ObjectAttributes->RootDirectory** handle was opened by file ID (using the FILE_OPEN_BY_FILE_ID flag) in a call to **ZwCreateFile**, and subsequently the file was opened by **ZwCreateFile** relative to this root-directory handle, **ZwQueryInformationFile** returns the relative path.
- However, if the user has **SeChangeNotifyPrivilege** (described in the Microsoft Windows SDK documentation), **ZwQueryInformationFile** returns the full path in all cases.
- If only the relative path is returned, the file name string will not begin with a backslash.
- If the full path and file name are returned, the string will begin with a single backslash, regardless of its location. Thus the file C:\dir1\dir2\filename.ext will appear as \dir1\dir2\filename.ext, while the file \server\share\dir1\dir2\filename.ext will appear as \server\share\dir1\dir2\filename.ext.

If **ZwQueryInformationFile** fails because of a buffer overflow, drivers that implement **FileNameInformation** should return as many WCHAR characters of the file name as will fit in the buffer and specify the full length that is required in the *FileNameLength* parameter of the FILE_NAME_INFORMATION structure. You should reissue the query by using the file name length so that you can retrieve the full file name. Drivers that do not follow this pattern might require a gradual increase in length until they retrieve the full file name. For more information about working with files, see Using Files in a Driver.

Callers of **ZwQueryInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtQueryInformationFile**" instead of "**ZwQueryInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

FILE_ACCESS_INFORMATION

FILE_ALIGNMENT_INFORMATION

FILE_ALL_INFORMATION

FILE_ATTRIBUTE_TAG_INFORMATION

FILE_BASIC_INFORMATION

FILE_EA_INFORMATION

FILE_INTERNAL_INFORMATION

FILE_IO_PRIORITY_HINT_INFORMATION

FILE_IS_REMOTE_DEVICE_INFORMATION

FILE_MODE_INFORMATION

FILE_NAME_INFORMATION

FILE_NETWORK_OPEN_INFORMATION

FILE_POSITION_INFORMATION

FILE_STANDARD_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateFile

ZwSetInformationFile

ZwQueryInformationResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationResourceManager** routine retrieves information about a specified resource manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwQueryInformationResourceManager(
  HANDLE                            ResourceManagerHandle,
  RESOURCEMANAGER_INFORMATION_CLASS ResourceManagerInformationClass,
  PVOID                             ResourceManagerInformation,
  ULONG                             ResourceManagerInformationLength,
  PULONG                            ReturnLength
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_QUERY_INFORMATION access to the object.

`ResourceManagerInformationClass`

A RESOURCEMANAGER_INFORMATION_CLASS-typed value that specifies the information to retrieve. This value must be **ResourceManagerBasicInformation**.

`ResourceManagerInformation`

A pointer to a caller-allocated RESOURCEMANAGER_BASIC_INFORMATION structure that receives information from **ZwQueryInformationResourceManager**.

`ResourceManagerInformationLength`

The length, in bytes, of the buffer that the *ResourceManagerInformation* parameter points to.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *ResourceManagerInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *ResourceManagerInformationClass* parameter's value is invalid.
STATUS_BUFFER_TOO_SMALL	The buffer size that the *ResourceManagerInformationLength* parameter specifies is smaller than the RESOURCEMANAGER_BASIC_INFORMATION structure.
STATUS_BUFFER_OVERFLOW	The buffer size that the *ResourceManagerInformationLength* parameter specifies is too small to receive all the variable-length information that is available.

STA **TUS** **_AC** **CES** **S_D** **ENI** **ED**	The caller does not have appropriate access to the resource manager object.

The routine might return other NTSTATUS values.

Remarks

For more information about the **ZwQueryInformationResourceManager** routine, see Creating a Resource Manager.

NtQueryInformationResourceManager and **ZwQueryInformationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

RESOURCEMANAGER_BASIC_INFORMATION

RESOURCEMANAGER_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateResourceManager

ZwOpenResourceManager

ZwRecoverResourceManager

ZwSetInformationResourceManager

ZwQueryInformationTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationTransaction** routine retrieves information about a specified transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwQueryInformationTransaction(
  HANDLE                        TransactionHandle,
  TRANSACTION_INFORMATION_CLASS TransactionInformationClass,
  PVOID                         TransactionInformation,
  ULONG                         TransactionInformationLength,
  PULONG                        ReturnLength
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_QUERY_INFORMATION access to the object.

`TransactionInformationClass`

A TRANSACTION_INFORMATION_CLASS-typed value that specifies the information to obtain. The value must be one of the following values:

- **TransactionBasicInformation**
- **TransactionPropertiesInformation**
- **TransactionEnlistmentInformation**

The **TransactionFullInformation** value is not used with **ZwQueryInformationTransaction**.

`TransactionInformation`

A pointer to a caller-allocated buffer that receives the information that the *TransactionInformationClass* parameter specifies. The buffer's structure type must be TRANSACTION_BASIC_INFORMATION, TRANSACTION_PROPERTIES_INFORMATION, or TRANSACTION_ENLISTMENTS_INFORMATION.

`TransactionInformationLength`

The length, in bytes, of the buffer that the *TransactionInformation* parameter points to, including the length of any additional array elements that the caller has allocated to receive information.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *TransactionInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The length of the buffer that is specified by the *TransactionInformationLength* parameter is incorrect.
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle*parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.

STA **TUS** **_BU** **FFE** **R_O** **VER** **FLO** **W**	The buffer that the *TransactionInformation*parameter specifies is too small.

The routine might return other NTSTATUS values.

Remarks

For more information about how to use **ZwQueryInformationTransaction**, see Creating a Transactional Client.

NtQueryInformationTransaction and **ZwQueryInformationTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_BASIC_INFORMATION

TRANSACTION_ENLISTMENTS_INFORMATION

TRANSACTION_INFORMATION_CLASS

TRANSACTION_PROPERTIES_INFORMATION

TmGetTransactionId

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

ZwSetInformationTransaction

ZwQueryInformationTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryInformationTransactionManager** routine retrieves information about a specified transaction manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwQueryInformationTransactionManager(
  HANDLE                              TransactionManagerHandle,
  TRANSACTIONMANAGER_INFORMATION_CLASS TransactionManagerInformationClass,
  PVOID                               TransactionManagerInformation,
  ULONG                               TransactionManagerInformationLength,
  PULONG                              ReturnLength
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_QUERY_INFORMATION access to the object.

`TransactionManagerInformationClass`

A TRANSACTIONMANAGER_INFORMATION_CLASS-typed enumeration value that specifies the information to be obtained. This value must be one of the following:

- **TransactionManagerBasicInformation**
- **TransactionManagerLogInformation**
- **TransactionManagerLogPathInformation**

The enumeration's **TransactionManagerOnlineProbeInformation** value is not used with **ZwQueryInformationTransactionManager**.

`TransactionManagerInformation`

A pointer to a caller-allocated buffer that receives the information that the *TransactionManagerInformationClass* parameter specifies. The buffer's structure type must be TRANSACTIONMANAGER_BASIC_INFORMATION, TRANSACTIONMANAGER_LOG_INFORMATION, TRANSACTIONMANAGER_LOGPATH_INFORMATION, or TRANSACTIONMANAGER_RECOVERY_INFORMATION.

`TransactionManagerInformationLength`

The length, in bytes, of the buffer that the *TransactionManagerInformation* parameter points to, including the length of any additional array elements that the caller has allocated to receive information.

`ReturnLength`

A pointer to a caller-allocated variable that receives the length, in bytes, of the information that KTM writes to the *TransactionManagerInformation* buffer. This parameter is optional and can be **NULL**.

Return Value

ZwQueryInformationTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionManagerInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The length of the buffer that is specified by the *TransactionManagerInformationLength*parameter is incorrect.
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction manager object.

STA **TUS** **_BU** **FFE** **R_T** **OO_** **SM** **ALL**	The buffer that is specified by the *TransactionManagerInformation*parameter is too small.

The routine might return other NTSTATUS values.

Remarks

For more information about how to use **ZwQueryInformationTransactionManager**, see Using Log Streams with KTM.

NtQueryInformationTransactionManager and **ZwQueryInformationTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTIONMANAGER_BASIC_INFORMATION

TRANSACTIONMANAGER_INFORMATION_CLASS

TRANSACTIONMANAGER_LOGPATH_INFORMATION

TRANSACTIONMANAGER_LOG_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRecoverTransactionManager

ZwRollforwardTransactionManager

ZwQueryKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryKey** routine provides information about the class of a registry key, and the number and sizes of its subkeys.

Syntax

```
NTSYSAPI NTSTATUS ZwQueryKey(
  HANDLE                KeyHandle,
  KEY_INFORMATION_CLASS KeyInformationClass,
  PVOID                 KeyInformation,
  ULONG                 Length,
  PULONG                ResultLength
);
```

Parameters

`KeyHandle`

Handle to the registry key to obtain information about. This handle is created by a successful call to ZwCreateKey or ZwOpenKey.

`KeyInformationClass`

Specifies a KEY_INFORMATION_CLASS value that determines the type of information returned in the *KeyInformation* buffer.

`KeyInformation`

Pointer to a caller-allocated buffer that receives the requested information.

`Length`

Specifies the size, in bytes, of the *KeyInformation* buffer.

`ResultLength`

Pointer to a variable that receives the size, in bytes, of the requested key information. If **ZwQueryKey** returns STATUS_SUCCESS, the variable contains the amount of data returned. If **ZwQueryKey** returns STATUS_BUFFER_OVERFLOW or STATUS_BUFFER_TOO_SMALL, you can use the value of the variable to determine the required buffer size.

Return Value

ZwQueryKey returns STATUS_SUCCESS on success, or the appropriate error code on failure. Possible error code values include:

RETURN CODE	DESCRIPTION

STATUS_BUFFER_OVERFLOW	The buffer supplied is too small, and only partial data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_BUFFER_TOO_SMALL	The buffer supplied is too small, and no data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_INVALID_PARAMETER	The *KeyInformationClass* parameter is not a valid KEY_INFORMATION_CLASS value.

Remarks

The*KeyHandle* passed to **ZwQueryKey** must have been opened with KEY_QUERY_VALUE access. This is accomplished by passing KEY_QUERY_VALUE, KEY_READ, or KEY_ALL_ACCESS as the *DesiredAccess* parameter to ZwCreateKey or ZwOpenKey.

> If *KeyInformationClass* is KeyNameInformation or KeyHandleTagsInformation, the KEY_QUERY_VALUE requirement does not apply.

ZwQueryKey can be used to obtain information that you can use to allocate buffers to hold registry data, such as the maximum size of a key's value entries or subkey names, or the number of subkeys. For example, you can call **ZwQueryKey**, use the returned information to allocate a buffer for a subkey, call ZwEnumerateKey to get the name of the subkey, and pass that name to an **Rtl*Xxx*Registry** routine.

For more information about working with registry keys, see Using the Registry in a Driver.

Note If the call to this function occurs in user mode, you should use the name "**NtQueryKey**" instead of "**ZwQueryKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

KEY_BASIC_INFORMATION

KEY_CACHED_INFORMATION

KEY_FULL_INFORMATION

KEY_INFORMATION_CLASS

KEY_NAME_INFORMATION

KEY_NODE_INFORMATION

KEY_VIRTUALIZATION_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwEnumerateKey

ZwOpenKey

ZwQuerySymbolicLinkObject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQuerySymbolicLinkObject** routine returns a Unicode string that contains the target of a symbolic link.

Syntax

```
NTSYSAPI NTSTATUS ZwQuerySymbolicLinkObject(
  HANDLE          LinkHandle,
  PUNICODE_STRING LinkTarget,
  PULONG          ReturnedLength
);
```

Parameters

`LinkHandle`

Handle to the symbolic-link object that you want to query. This handle is created by a successful call to ZwOpenSymbolicLinkObject.

`LinkTarget`

Pointer to an initialized Unicode string that receives the target of the symbolic link.

`ReturnedLength`

contains the maximum number of bytes to copy into the Unicode string at *LinkTarget*. On output, the unsigned long integer contains the length of the Unicode string naming the target of the symbolic link.

Return Value

ZwQuerySymbolicLinkObject returns either STATUS_SUCCESS to indicate the routine completed without error or STATUS_BUFFER_TOO_SMALL if the Unicode string provided at *LinkTarget* is too small to hold the returned string.

Remarks

Before calling this routine, driver writers must ensure that the Unicode string at *LinkTarget*has been properly initialized and a buffer for the string has been allocated. The **MaximumLength** and **Buffer** members of the Unicode string must be set before calling **ZwQuerySymbolicLinkObject** or the call will fail.

If **ZwQuerySymbolicLinkObject** returns STATUS_BUFFER_TOO_SMALL drivers should examine the value returned at *ReturnedLength*. The number returned in this variable indicates the maximum length that the Unicode string for the target of the symbolic link.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtQuerySymbolicLinkObject**" instead of "**ZwQuerySymbolicLinkObject**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System

Services Routines.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwOpenSymbolicLinkObject

ZwQueryValueKey function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwQueryValueKey** routine returns a value entry for a registry key.

Syntax

```
NTSYSAPI NTSTATUS ZwQueryValueKey(
  HANDLE                      KeyHandle,
  PUNICODE_STRING             ValueName,
  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
  PVOID                       KeyValueInformation,
  ULONG                       Length,
  PULONG                      ResultLength
);
```

Parameters

`KeyHandle`

Handle to the key to read value entries from. This handle is created by a successful call to ZwCreateKey or ZwOpenKey.

`ValueName`

Pointer to the name of the value entry to obtain data for.

`KeyValueInformationClass`

A KEY_VALUE_INFORMATION_CLASS value that determines the type of information returned in the *KeyValueInformation* buffer.

`KeyValueInformation`

Pointer to a caller-allocated buffer that receives the requested information.

`Length`

Specifies the size, in bytes, of the *KeyValueInformation* buffer.

`ResultLength`

Pointer to a variable that receives the size, in bytes, of the key information. If the **ZwQueryValueKey** routine returns STATUS_SUCCESS, callers can use the value of this variable to determine the amount of data returned. If the routine returns STATUS_BUFFER_OVERFLOW or STATUS_BUFFER_TOO_SMALL, callers can use the value of this variable to determine the size of buffer required to hold the key information.

Return Value

ZwQueryValueKey returns STATUS_SUCCESS on success, or the appropriate error code on failure. Possible error code values include:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_NAME_NOT_FOUND	The registry value was not available.
STATUS_BUFFER_OVERFLOW	The buffer supplied is too small, and only partial data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_BUFFER_TOO_SMALL	The buffer supplied is too small, and no data has been written to the buffer. **ResultLength* is set to the minimum size required to hold the requested information.
STATUS_INVALID_PARAMETER	The *KeyInformationClass* parameter is not a valid **KEY_VALUE_INFORMATION_CLASS** value. **Warning** If you specify **KeyValueBasicInformation** for *KeyValueInformationClass*, Windows 98 and Windows Me return STATUS_SUCCESS even if the registry key or value name does not exist.

Remarks

The*KeyHandle* passed to **ZwQueryValueKey** must have been opened with KEY_QUERY_VALUE access. This is accomplished by passing KEY_QUERY_VALUE, KEY_READ, or KEY_ALL_ACCESS as the *DesiredAccess* parameter to ZwCreateKey or ZwOpenKey.

For more information about working with registry keys, see Using the Registry in a Driver.

Note If the call to this function occurs in user mode, you should use the name "**NtQueryValueKey**" instead of "**ZwQueryValueKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System

Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

KEY_VALUE_BASIC_INFORMATION

KEY_VALUE_FULL_INFORMATION

KEY_VALUE_INFORMATION_CLASS

KEY_VALUE_PARTIAL_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateKey

ZwEnumerateValueKey

ZwOpenKey

ZwReadFile function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwReadFile** routine reads data from an open file.

Syntax

```
NTSYSAPI NTSTATUS ZwReadFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to ZwCreateFile or ZwOpenFile.

`Event`

Optionally, a handle to an event object to set to the signaled state after the read operation completes. Device and intermediate drivers should set this parameter to **NULL**.

`ApcRoutine`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`ApcContext`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested read operation. The **Information** member receives the number of bytes actually read from the file.

`Buffer`

Pointer to a caller-allocated buffer that receives the data read from the file.

`Length`

The size, in bytes, of the buffer pointed to by *Buffer*.

`ByteOffset`

Pointer to a variable that specifies the starting byte offset in the file where the read operation will begin. If an attempt is made to read beyond the end of the file, **ZwReadFile** returns an error.

If the call to ZwCreateFile set either of the **CreateOptions** flags FILE_SYNCHRONOUS_IO_ALERT or

FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **ZwReadFile** can specify that the current file position offset be used instead of an explicit **ByteOffset** value. This specification can be made by using one of the following methods:

- Specify a pointer to a LARGE_INTEGER value with the **HighPart** member set to -1 and the **LowPart** member set to the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a **NULL** pointer for *ByteOffset*.

ZwReadFile updates the current file position by adding the number of bytes read when it completes the read operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **ZwReadFile**. Doing this automatically changes the current file position to that *ByteOffset* value, performs the read operation, and then updates the position according to the number of bytes actually read. This technique gives the caller atomic seek-and-read service.

`Key`

Device and intermediate drivers should set this pointer to **NULL**.

Return Value

ZwReadFile returns either STATUS_SUCCESS or the appropriate NTSTATUS error code.

Remarks

Callers of **ZwReadFile** must have already called ZwCreateFile with the FILE_READ_DATA or GENERIC_READ value set in the *DesiredAccess* parameter.

If the preceding call to ZwCreateFile set the FILE_NO_INTERMEDIATE_BUFFERING flag in the *CreateOptions* parameter to **ZwCreateFile**, the *Length* and *ByteOffset* parameters to **ZwReadFile** must be multiples of the sector size. For more information, see **ZwCreateFile**.

ZwReadFile begins reading from the given *ByteOffset* or the current file position into the given *Buffer*. It terminates the read operation under one of the following conditions:

- The buffer is full because the number of bytes specified by the *Length* parameter has been read. Therefore, no more data can be placed into the buffer without an overflow.
- The end of file is reached during the read operation, so there is no more data in the file to be transferred into the buffer.

If the caller opened the file with the SYNCHRONIZE flag set in *DesiredAccess*, the calling thread can synchronize to the completion of the read operation by waiting on the file handle, *FileHandle*. The handle is signaled each time that an I/O operation that was issued on the handle completes. However, the caller must not wait on a handle that was opened for synchronous file access (FILE_SYNCHRONOUS_IO_NONALERT or FILE_SYNCHRONOUS_IO_ALERT). In this case, **ZwReadFile** waits on behalf of the caller and does not return until the read operation is complete. The caller can safely wait on the file handle only if all three of the following conditions are met:

- The handle was opened for asynchronous access (that is, neither FILE_SYNCHRONOUS_IO_**XXX** flag was specified).
- The handle is being used for only one I/O operation at a time.
- **ZwReadFile** returned STATUS_PENDING.

A driver should call **ZwReadFile** in the context of the system process if any of the following conditions exist:

- The driver created the file handle that it passes to **ZwReadFile**.
- **ZwReadFile** will notify the driver of I/O completion by means of an event that the driver created.
- **ZwReadFile** will notify the driver of I/O completion by means of an APC callback routine that the driver passes to **ZwReadFile**.

File and event handles are valid only in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **ZwReadFile** in the context of the system process rather than the context of the process that the driver is in.

Likewise, **ZwReadFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread that issues the I/O request. If the driver calls **ZwReadFile** in the context of a process other than the system one, the APC could be delayed indefinitely, or it might not fire at all.

For more information about working with files, see Using Files in a Driver.

Callers of **ZwReadFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "NtReadFile" instead of "**ZwReadFile**".

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, BufAfterReqCompletedIntIoctlA, BufAfterReqCompletedIoctlA, BufAfterReqCompletedReadA, BufAfterReqCompletedWriteA, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

ZwCreateFile

ZwQueryInformationFile

ZwSetInformationFile

ZwWriteFile

ZwReadOnlyEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwReadOnlyEnlistment** routine sets a specified enlistment to be read-only.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwReadOnlyEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwReadOnlyEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state. For example, an enlistment cannot be read-only if the ENLISTMENT_SUPERIOR flag was specified to ZwCreateEnlistment.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

Your component cannot call **ZwReadOnlyEnlistment** for a superior enlistment.

For more information about **ZwReadOnlyEnlistment**, see Creating a Read-Only Enlistment.

NtReadOnlyEnlistment and **ZwReadOnlyEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwRecoverEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverEnlistment** routine initiates a recovery operation for the transaction that is associated with a specified enlistment.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRecoverEnlistment(
  HANDLE EnlistmentHandle,
  PVOID  EnlistmentKey
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_RECOVER access to the object.

`EnlistmentKey`

A pointer to the enlistment key value that the resource manager previously specified as the *EnlistmentKey* parameter to ZwCreateEnlistment. This parameter is optional and can be **NULL** if the resource manager did not provide an enlistment key when it called **ZwCreateEnlistment**.

Return Value

ZwRecoverEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.

STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_TRANSACTION_NOT_EQUEST_NOT_VALID	The transaction or its enlistment is not in the correct state.
STATUS_PENDING	KTM has added a commit, rollback, or in-doubt notification to the resource manager's notification queue, and the resource manager is not using a ResourceManagerNotification callback routine.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

After a resource manager calls **ZwRecoverEnlistment**, KTM delivers either a TRANSACTION_NOTIFY_COMMIT, TRANSACTION_NOTIFY_ROLLBACK, or TRANSACTION_NOTIFY_INDOUBT notification for the enlistment. Therefore, unless the resource manager is using a ResourceManagerNotification callback routine, it must immediately call ZwGetNotificationResourceManager if **ZwRecoverEnlistment** returns STATUS_SUCCESS or STATUS_PENDING.

For more information about **ZwRecoverEnlistment**, see Handling Recovery Operations.

NtRecoverEnlistment and **ZwRecoverEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine

can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ResourceManagerNotification

TmRecoverEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwGetNotificationResourceManager

ZwOpenEnlistment

ZwRecoverResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverResourceManager** routine tries to recover the transaction that is associated with each enlistment of a specified resource manager object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRecoverResourceManager(
  HANDLE ResourceManagerHandle
);
```

Parameters

`ResourceManagerHandle`

A handle to a resource manager object that was obtained by a previous call to ZwCreateResourceManager or ZwOpenResourceManager. The handle must have RESOURCEMANAGER_RECOVER access to the object.

Return Value

ZwRecoverResourceManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a resource manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STATUS_TRANSACTION_NOT_ENLISTED	The transaction manager that is associated with the specified resource manager is not available.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the resource manager object.

The routine might return other NTSTATUS values.

Remarks

When a resource manager calls **ZwRecoverResourceManager**, KTM sends a TRANSACTION_NOTIFY_RECOVER notification to the resource manager for each of the resource manager's enlistments. (If the enlistment is a superior enlistment, KTM sends TRANSACTION_NOTIFY_RECOVER_QUERY instead of TRANSACTION_NOTIFY_RECOVER.)

The final notification that KTM sends to the resource manager is TRANSACTION_NOTIFY_LAST_RECOVER.

For more information about **ZwRecoverResourceManager**, see Handling Recovery Operations.

NtRecoverResourceManager and **ZwRecoverResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)

Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

TmRecoverResourceManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateResourceManager

ZwOpenResourceManager

ZwQueryInformationResourceManager

ZwSetInformationResourceManager

ZwRecoverTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRecoverTransactionManager** routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRecoverTransactionManager(
  HANDLE TransactionManagerHandle
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_RECOVER access to the object.

Return Value

ZwRecoverTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the transaction manager object.
STA TUS _TM _VO LAT ILE	The transaction manager object is volatile and, therefore, its transactions cannot be recovered.
STA TUS _UN SUC CES SFU L	The transaction manager object is not in a state that allows recovery.

The routine might return other NTSTATUS values.

Remarks

The **ZwRecoverTransactionManager** routine tries to reconstruct the state of the transaction manager object, including all transactions, enlistments, and resource managers, from the recovery information that is in the object's log stream, beginning at the last restart area that KTM created and ending at the stream's end.

To recover up to a specific virtual time, use the ZwRollforwardTransactionManager routine instead of **ZwRecoverTransactionManager**.

Your TPS component must call **ZwRecoverTransactionManager** after it has called ZwCreateTransactionManager or ZwOpenTransactionManager.

For more information about recovery operations, see Handling Recovery Operations.

NtRecoverTransactionManager and **ZwRecoverTransactionManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.

Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRecoverTransactionManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRollforwardTransactionManager

ZwRollbackComplete function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackComplete** routine notifies KTM that the calling resource manager has finished rolling back a transaction's data.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRollbackComplete(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwRollbackComplete returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager must call **ZwRollbackComplete** after it has finished servicing a TRANSACTION_NOTIFY_ROLLBACK notification.

For more information about **ZwRollbackComplete**, see Handling Rollback Operations.

NtRollbackComplete and **ZwRollbackComplete** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRollbackComplete

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwRollbackEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackEnlistment** routine rolls back the transaction that is associated with a specified enlistment.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRollbackEnlistment(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwRollbackEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_R EQU EST _NO T_V ALI D	The enlistment cannot be rolled back.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call **ZwRollbackEnlistment** to roll back a transaction at any time before it calls ZwPrepareComplete.

For more information about **ZwRollbackEnlistment**, see Handling Rollback Operations.

NtRollbackEnlistment and **ZwRollbackEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRollbackEnlistment

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwPrepareComplete

ZwRollbackTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollbackTransaction** routine initiates a rollback operation for a specified transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRollbackTransaction(
  HANDLE  TransactionHandle,
  BOOLEAN Wait
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_ROLLBACK access to the object.

`Wait`

A Boolean value that the caller sets to **TRUE** for synchronous operation or **FALSE** for asynchronous operation. If this parameter is set to **TRUE**, the call does not return until the rollback operation is complete.

Return Value

ZwRollbackTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle* parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.

STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.
STATUS_TRANSACTION_ALREADY_COMMITTED	The transaction cannot be rolled back because it has already been committed.
STATUS_TRANSACTION_NOT_ROLLBACK	The transaction has not been committed but its current state does not permit rollback.
STATUS_PENDING	Rollback notifications have been queued to resource managers, and the caller specified **FALSE** for the *Wait* parameter.

The routine might return other NTSTATUS values.

Remarks

For more information about **ZwRollbackTransaction**, see Handling Rollback Operations.

NtRollbackEnlistment and **ZwRollbackEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwCommitTransaction

ZwCreateTransaction

ZwOpenTransaction

ZwQueryInformationTransaction

ZwRollforwardTransactionManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwRollforwardTransactionManager** routine initiates recovery operations for all of the in-progress transactions that are assigned to a specified transaction manager.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwRollforwardTransactionManager(
  HANDLE         TransactionManagerHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`TransactionManagerHandle`

A handle to a transaction manager object that was obtained by a previous call to ZwCreateTransactionManager or ZwOpenTransactionManager. The handle must have TRANSACTIONMANAGER_RECOVER access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**. For more information about this parameter, see the following Remarks section.

Return Value

ZwRollforwardTransactionManager returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to a transaction manager object.

STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction manager object.
STATUS_TM_VOLATILE	The specified transaction manager object does not have a log file, so recovery is not available.
STATUS_UNSUCCESSFUL	The specified transaction manager object is in an unexpected state.

The routine might return other NTSTATUS values.

Remarks

The **ZwRollforwardTransactionManager** routine recovers all logged activity that KTM finds in the transaction manager's log file, up to and including the virtual clock value that the *TmVirtualClock* parameter supplies.

Your component can traverse the log file incrementally by calling **ZwRollforwardTransactionManager** repetitively and setting the *VirtualClock* parameter to a higher value before each call.

If the *TmVirtualClock* parameter is **NULL**, calling **ZwRollforwardTransactionManager** is equivalent to calling ZwRecoverTransactionManager.

For more information about recovery operations, see Handling Recovery Operations.

NtRollforwardTransactionManager and **ZwRollforwardTransactionManager** are two versions of the same Windows Native System Services routine. The **NtRollforwardTransactionManager** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwRollforwardTransactionManager** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later versions of Windows.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmRecoverTransactionManager

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransactionManager

ZwOpenTransactionManager

ZwRecoverTransactionManager

ZwSetInformationEnlistment function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationEnlistment** routine sets information for a specified enlistment object.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwSetInformationEnlistment(
  HANDLE                      EnlistmentHandle,
  ENLISTMENT_INFORMATION_CLASS EnlistmentInformationClass,
  PVOID                       EnlistmentInformation,
  ULONG                       EnlistmentInformationLength
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SET_INFORMATION access to the object.

`EnlistmentInformationClass`

A ENLISTMENT_INFORMATION_CLASS-typed enumeration value that specifies the type of information to be set. This value must be **EnlistmentRecoveryInformation**.

The enumeration's **EnlistmentBasicInformation** and **EnlistmentFullInformation** values are not used with **ZwSetInformationEnlistment**.

`EnlistmentInformation`

A pointer to a caller-allocated buffer that contains caller-defined recovery information for the enlistment.

`EnlistmentInformationLength`

The length, in bytes, of the buffer that the *EnlistmentInformation* parameter points to.

Return Value

ZwSetInformationEnlistment returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION

STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.
STATUS_INVALID_INFO_CLASS	The *EnlistmentInformationClass* parameter's value is invalid.
STATUS_INFO_LENGTH_MISMATCH	The *EnlistmentInformationLength* parameter's value is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can use the **ZwSetInformationEnlistment** routine to set recovery information for an enlistment. KTM writes the recovery information to the log stream, and the resource manager can call

ZwQueryInformationEnlistment to read this information from the log stream at any time.

Each subsequent call to **ZwSetInformationEnlistment** deletes the recovery information that the previous call specified before it stores the new recovery information.

For more information about how to use **ZwSetInformationEnlistment**, see Using Log Streams with KTM.

NtSetInformationEnlistment and **ZwSetInformationEnlistment** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

ENLISTMENT_INFORMATION_CLASS

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwQueryInformationEnlistment

ZwSetInformationFile function

1/25/2019 • 4 minutes to read • Edit Online

The **ZwSetInformationFile** routine changes various kinds of information about a file object.

Syntax

```
NTSYSAPI NTSTATUS ZwSetInformationFile(
  HANDLE                 FileHandle,
  PIO_STATUS_BLOCK       IoStatusBlock,
  PVOID                  FileInformation,
  ULONG                  Length,
  FILE_INFORMATION_CLASS FileInformationClass
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to ZwCreateFile or ZwOpenFile.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested operation. The **Information** member receives the number of bytes set on the file.

`FileInformation`

Pointer to a buffer that contains the information to set for the file. The particular structure in this buffer is determined by the *FileInformationClass* parameter. Setting any member of the structure to zero tells **ZwSetInformationFile** to leave the current information about the file for that member unchanged.

`Length`

The size, in bytes, of the *FileInformation* buffer.

`FileInformationClass`

The type of information, supplied in the buffer pointed to by *FileInformation*, to set for the file. Device and intermediate drivers can specify any of the following FILE_INFORMATION_CLASS values.

***FILEINFORMATIONCLASS* VALUE**	**MEANING**
FileBasicInformation	Change the information that is supplied in a FILE_BASIC_INFORMATION structure. The caller must have opened the file with the FILE_WRITE_ATTRIBUTES flag set in the *DesiredAccess* parameter.
FileDispositionInformation	Usually, sets the **DeleteFile** member of a FILE_DISPOSITION_INFORMATION to **TRUE**, so the file can be deleted when ZwClose is called to release the last open handle to the file object. The caller must have opened the file with the DELETE flag set in the *DesiredAccess* parameter.

FileEndOfFileInformation	Change the current end-of-file information, supplied in a FILE_END_OF_FILE_INFORMATION structure. The operation can either truncate or extend the file. The caller must have opened the file with the FILE_WRITE_DATA flag set in the *DesiredAccess* parameter.
FileIoPriorityHintInformation	Change the current default IRP priority hint for the file handle. The new value is supplied in a FILE_IO_PRIORITY_HINT_INFORMATION structure. **Note** This structure must be 8-byte aligned.
FileLinkInformation	Create a hard link to an existing file, which is specified in a FILE_LINK_INFORMATION structure. Not all file systems support hard links; for example NTFS does while FAT does not.
FilePositionInformation	Change the current file information, which is stored in a FILE_POSITION_INFORMATION structure.
FileRenameInformation	Change the current file name, which is supplied in a FILE_RENAME_INFORMATION structure. The caller must have DELETE access to the file.
FileShortNameInformation	Change the current short file name, which is supplied in a FILE_NAME_INFORMATION structure. The file must be on an NTFS volume, and the caller must have opened the file with the *DesiredAccess* DELETE flag set in the *DesiredAccess* parameter.
FileValidDataLengthInformation	Change the current valid data length for the file, which is supplied in a FILE_VALID_DATA_LENGTH_INFORMATION structure. The file must be on an NTFS volume, and the caller must have opened the file with the FILE_WRITE_DATA flag set in the *DesiredAccess* parameter. Non-administrators and remote users must have the **SeManageVolumePrivilege** privilege.
FileReplaceCompletionInformation	Change or remove the I/O completion port for the specified file handle. The caller supplies a pointer to a FILE_COMPLETION_INFORMATION structure that specifies a port handle and a completion key. If the port handle is non-NULL, this handle specifies a new I/O completion port to associate with the file handle. To remove the I/O completion port associated with the file handle, set the port handle in the structure to NULL. To get a port handle, a user-mode caller can call the CreateIoCompletionPort function.

Return Value

ZwSetInformationFile returns STATUS_SUCCESS or an appropriate error status.

Remarks

ZwSetInformationFile changes information about a file. It ignores any member of a **FILE_*XXX*_INFORMATION** structure that is not supported by a particular device or file system.

If you set *FileInformationClass* to **FileDispositionInformation**, you can subsequently pass *FileHandle* to ZwClose

but not to any other **Zw*Xxx*File** routine. Because **FileDispositionInformation** causes the file to be marked for deletion, it is a programming error to attempt any subsequent operation on the handle other than closing it.

If you set *FileInformationClass* to **FilePositionInformation**, and the preceding call to **ZwCreateFile** included the FILE_NO_INTERMEDIATE_BUFFERING flag in the *CreateOptions* parameter, certain restrictions on the **CurrentByteOffset** member of the FILE_POSITION_INFORMATION structure are enforced. For more information, see ZwCreateFile.

If you set *FileInformationClass* to **FileEndOfFileInformation**, and the **EndOfFile** member of FILE_END_OF_FILE_INFORMATION specifies an offset beyond the current end-of-file mark, **ZwSetInformationFile** extends the file and pads the extension with zeros.

For more information about working with files, see Using Files in a Driver.

Callers of **ZwSetInformationFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "**NtSetInformationFile**" instead of "**ZwSetInformationFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

CreateIoCompletionPort

FILE_BASIC_INFORMATION

FILE_COMPLETION_INFORMATION

FILE_DISPOSITION_INFORMATION

FILE_END_OF_FILE_INFORMATION

FILE_IO_PRIORITY_HINT_INFORMATION

FILE_LINK_INFORMATION

FILE_NAME_INFORMATION

FILE_POSITION_INFORMATION

FILE_RENAME_INFORMATION

FILE_VALID_DATA_LENGTH_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateFile

ZwOpenFile

ZwQueryInformationFile

ZwSetInformationResourceManager function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationResourceManager** routine is not used.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwSetInformationResourceManager(
  HANDLE                          ResourceManagerHandle,
  RESOURCEMANAGER_INFORMATION_CLASS ResourceManagerInformationClass,
  PVOID                           ResourceManagerInformation,
  ULONG                           ResourceManagerInformationLength
);
```

Parameters

`ResourceManagerHandle`

`ResourceManagerInformationClass`

`ResourceManagerInformation`

`ResourceManagerInformationLength`

Return Value

This function does not return a value.

Remarks

NtSetInformationResourceManager and **ZwSetInformationResourceManager** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Target Platform	Windows
Header	wdm.h

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwSetInformationTransaction function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSetInformationTransaction** routine sets information for a specified transaction.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwSetInformationTransaction(
  HANDLE                      TransactionHandle,
  TRANSACTION_INFORMATION_CLASS TransactionInformationClass,
  PVOID                       TransactionInformation,
  ULONG                       TransactionInformationLength
);
```

Parameters

`TransactionHandle`

A handle to a transaction object that was obtained by a previous call to ZwCreateTransaction or ZwOpenTransaction. The handle must have TRANSACTION_SET_INFORMATION access to the object.

`TransactionInformationClass`

A TRANSACTION_INFORMATION_CLASS-typed value that specifies the type of information to set. The value must be **TransactionPropertiesInformation**.

`TransactionInformation`

A pointer to a caller-allocated buffer that contains the information to set. The buffer's structure type must be TRANSACTION_PROPERTIES_INFORMATION.

`TransactionInformationLength`

The length, in bytes, of the buffer that the *TransactionInformation* parameter points to.

Return Value

ZwSetInformationTransaction returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_INVALID_INFO_CLASS	The *TransactionInformationClass* parameter's value is invalid.

STATUS_OBJECT_TYPE_MISMATCH	The handle that the *TransactionHandle* parameter specifies is not a handle to a transaction object.
STATUS_INVALID_HANDLE	An object handle is invalid.
STATUS_ACCESS_DENIED	The caller does not have appropriate access to the transaction object.
STATUS_INFO_LENGTH_MISMATCH	The *TransactionInformationLength*parameter's value is invalid.
STATUS_INVALID_PARAMETER	The contents of the buffer that the *TransactionInformation* buffer specifies is invalid.

The routine might return other NTSTATUS values.

Remarks

For more information about **ZwSetInformationTransaction**, see Creating a Transactional Client.

NtSetInformationTransaction and **ZwSetInformationTransaction** are two versions of the same Windows Native System Services routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TRANSACTION_INFORMATION_CLASS

TRANSACTION_PROPERTIES_INFORMATION

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateTransaction

ZwOpenTransaction

ZwQueryInformationTransaction

ZwSetValueKey function

1/25/2019 • 3 minutes to read • Edit Online

The **ZwSetValueKey** routine creates or replaces a registry key's value entry.

Syntax

```
NTSYSAPI NTSTATUS ZwSetValueKey(
  HANDLE          KeyHandle,
  PUNICODE_STRING ValueName,
  ULONG           TitleIndex,
  ULONG           Type,
  PVOID           Data,
  ULONG           DataSize
);
```

Parameters

`KeyHandle`

Handle to the registry key to write a value entry for. This handle is created by a successful call to ZwCreateKey or ZwOpenKey.

`ValueName`

Pointer to the name of the value entry for which the data is to be written. This parameter can be a **NULL** pointer if the value entry has no name. If a name string is specified and the given name is not unique relative to its containing key, the data for an existing value entry is replaced.

`TitleIndex`

This parameter is reserved. Device and intermediate drivers should set this parameter to zero.

`Type`

One of the following system-defined types of data to write.

***TYPE* VALUE**	**MEANING**
REG_BINARY	Binary data in any form.
REG_DWORD	A 4-byte numerical value.
REG_DWORD_LITTLE_ENDIAN	A 4-byte numerical value with the least significant byte at the lowest address. Identical to REG_DWORD.
REG_DWORD_BIG_ENDIAN	A 4-byte numerical value with the least significant byte at the highest address.
REG_EXPAND_SZ	A null-terminated Unicode string that contains unexpanded references to environment variables, such as "%PATH%".

REG_LINK	A Unicode string that names a symbolic link. This type is irrelevant to device and intermediate drivers.
REG_MULTI_SZ	An array of null-terminated strings, terminated by another zero.
REG_NONE	Data with no particular type.
REG_SZ	A null-terminated Unicode string.
REG_RESOURCE_LIST	A device driver's list of hardware resources, used by the driver or one of the physical devices it controls, in the **\ResourceMap** tree
REG_RESOURCE_REQUIREMENTS_LIST	A device driver's list of possible hardware resources it or one of the physical devices it controls can use, from which the system writes a subset into the **\ResourceMap** tree
REG_FULL_RESOURCE_DESCRIPTOR	A list of hardware resources that a physical device is using, detected and written into the **\HardwareDescription** tree by the system

Note Device drivers should not attempt to call **ZwSetValueKey** to explicitly write value entries in a subkey of the **\Registry...\ResourceMap** key. Only the system can write value entries to the **\Registry...\HardwareDescription** tree.

`Data`

Pointer to a caller-allocated buffer that contains the data for the value entry.

`DataSize`

Specifies the size, in bytes, of the *Data* buffer. If *Type* is REG_*XXX*_SZ, this value must include space for any terminating zeros.

Return Value

ZwSetValueKey returns an NTSTATUS value. Possible return values include:

Remarks

The*KeyHandle* passed to **ZwSetValueKey** must have been opened with the KEY_SET_VALUE *DesiredAccess* flag set for this call to succeed. For a description of possible values for *DesiredAccess*, see ZwCreateKey.

If the given key has no existing value entry with a name matching the given *ValueName*, **ZwSetValueKey** creates a new value entry with the given name. If a matching value entry name exists, this routine overwrites the original value entry for the given *ValueName*. Thus, **ZwSetValueKey** preserves a unique name for each value entry of any particular key. While each value entry name must be unique to its containing key, many different keys in the registry can have value entries with the same names.

For more information about working with registry keys, see Using the Registry in a Driver.

Note If the call to this function occurs in user mode, you should use the name "NtSetValueKey" instead of "**ZwSetValueKey**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the

relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	IrqlZwPassive, PowerIrpDDis, ZwRegistryCreate, ZwRegistryOpen, HwStorPortProhibitedDDIs, ZwRegistryCreate(storport), ZwRegistryOpen(storport)

See Also

HalAssignSlotResources

IoAssignResources

IoQueryDeviceDescription

IoReportResourceUsage

Using Nt and Zw Versions of the Native System Services Routines

ZwClose

ZwCreateKey

ZwFlushKey

ZwOpenKey

ZwSinglePhaseReject function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwSinglePhaseReject** routine informs KTM that the calling resource manager will not support single-phase commit operations for a specified enlistment.

Syntax

```
NTSYSCALLAPI NTSTATUS ZwSinglePhaseReject(
  HANDLE         EnlistmentHandle,
  PLARGE_INTEGER TmVirtualClock
);
```

Parameters

`EnlistmentHandle`

A handle to an enlistment object that was obtained by a previous call to ZwCreateEnlistment or ZwOpenEnlistment. The handle must have ENLISTMENT_SUBORDINATE_RIGHTS access to the object.

`TmVirtualClock`

A pointer to a virtual clock value. This parameter is optional and can be **NULL**.

Return Value

ZwSinglePhaseReject returns STATUS_SUCCESS if the operation succeeds. Otherwise, this routine might return one of the following values:

RETURN CODE	DESCRIPTION
STATUS_OBJECT_TYPE_MISMATCH	The specified handle is not a handle to an enlistment object.
STATUS_INVALID_HANDLE	The object handle is invalid.

STA TUS _TR AN SAC TIO N_N OT_ REQ UES TED	The transaction or its enlistment is not in the correct state.
STA TUS _AC CES S_D ENI ED	The caller does not have appropriate access to the enlistment object.

The routine might return other NTSTATUS values.

Remarks

A resource manager can call **ZwSinglePhaseReject** when it receives a TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notification.

For more information about **ZwSinglePhaseReject**, see Handling Commit Operations.

NtSinglePhaseReject and **ZwSinglePhaseReject** are two versions of the same Windows Native System Services routine. The **NtSinglePhaseReject** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwSinglePhaseReject** routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available in Windows Vista and later operating system versions.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe

IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

TmSinglePhaseReject

Using Nt and Zw Versions of the Native System Services Routines

ZwCreateEnlistment

ZwOpenEnlistment

ZwUnloadDriver function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwUnloadDriver** routine unloads a driver from the system.
*Use this routine with extreme caution. (See the following***Remarks***section.)*

Syntax

```
NTSYSAPI NTSTATUS ZwUnloadDriver(
  PUNICODE_STRING DriverServiceName
);
```

Parameters

`DriverServiceName`

Pointer to a counted Unicode string that specifies a path to the driver's registry key, **\Registry\Machine\System\CurrentControlSet\Services*DriverName*, where *DriverName* is the name of the driver.**

Return Value

ZwUnloadDriver returns STATUS_SUCCESS or an error NTSTATUS value such as STATUS_INVALID_DEVICE_REQUEST.

Remarks

ZwUnloadDriver dynamically unloads a device or file system driver from the currently running system.

Note that a file system filter driver cannot safely be unloaded from a running system. Thus a filter should only use
ZwUnloadDriverfor debugging purposes. It should not call this routine in a retail version of the filter.

If *DriverName* is the name of a PnP device driver, ZwUnloadDriver returns STATUS_INVALID_DEVICE_REQUEST and does not unload the driver.

A minifilter should use FltUnloadFilter instead of ZwUnloadDriver to unload a supporting minifilter.

> **Note If the call to the ZwUnloadDriver function occurs in user mode, you should use the name "NtUnloadDriver" instead of "ZwUnloadDriver".**

For calls from kernel-mode drivers, the Nt*Xxx* and Zw*Xxx* versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the Nt*Xxx* and Zw*Xxx* versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	**Available in Windows XP and later versions of Windows.**
Target Platform	**Universal**
Header	**wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)**
Library	**NtosKrnl.lib**
DLL	**NtosKrnl.exe**
IRQL	**PASSIVE_LEVEL**
DDI compliance rules	**PowerIrpDDis, HwStorPortProhibitedDDIs**

See Also

FltUnloadFilter

RtlInitUnicodeString

UNICODE_STRING

Using Nt and Zw Versions of the Native System Services Routines

ZwLoadDriver

ZwUnmapViewOfSection function

1/25/2019 • 2 minutes to read • Edit Online

The **ZwUnmapViewOfSection** routine unmaps a view of a section from the virtual address space of a subject process.

Syntax

```
NTSYSAPI NTSTATUS ZwUnmapViewOfSection(
  HANDLE ProcessHandle,
  PVOID  BaseAddress
);
```

Parameters

`ProcessHandle`

Handle to a process object that was previously passed to ZwMapViewOfSection.

`BaseAddress`

Pointer to the base virtual address of the view to unmap. This value can be any virtual address within the view.

Return Value

ZwUnmapViewOfSection returns an NTSTATUS value. Possible return values include:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully performed the requested operation.
STATUS_ACCESS_DENIED	The caller does not have access rights to the process object or to the base virtual address of the view.

Remarks

This routine unmaps the entire view of the section that contains *BaseAddress* from the virtual address space of the specified process—even if *BaseAddress* does not point to the beginning of the view.

On return from **ZwUnmapViewOfSection**, the virtual-address region occupied by the view is no longer reserved and is available to map other views or private pages. If the view was also the last reference to the underlying

section, all committed pages in the section are decommitted, and the section is deleted.

> **Note** If the call to this function occurs in user mode, you should use the name "NtUnmapViewOfSection" instead of "**ZwUnmapViewOfSection**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

Using Nt and Zw Versions of the Native System Services Routines

ZwMapViewOfSection

ZwOpenSection

ZwWriteFile function

1/25/2019 • 5 minutes to read • Edit Online

The **ZwWriteFile** routine writes data to an open file.

Syntax

```
NTSYSAPI NTSTATUS ZwWriteFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

Parameters

`FileHandle`

Handle to the file object. This handle is created by a successful call to ZwCreateFile or ZwOpenFile.

`Event`

Optionally, a handle to an event object to set to the signaled state after the write operation completes. Device and intermediate drivers should set this parameter to **NULL**.

`ApcRoutine`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`ApcContext`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`IoStatusBlock`

Pointer to an IO_STATUS_BLOCK structure that receives the final completion status and information about the requested write operation. The **Information** member receives the number of bytes actually written to the file.

`Buffer`

Pointer to a caller-allocated buffer that contains the data to write to the file.

`Length`

The size, in bytes, of the buffer pointed to by *Buffer*.

`ByteOffset`

Pointer to a variable that specifies the starting byte offset in the file for beginning the write operation. If *Length* and *ByteOffset* specify a write operation past the current end-of-file mark, **ZwWriteFile** automatically extends the file and updates the end-of-file mark; any bytes that are not explicitly written between such old and new end-of-file

marks are defined to be zero.

If the call to ZwCreateFile set only the *DesiredAccess* flag FILE_APPEND_DATA, *ByteOffset* is ignored. Data in the given *Buffer*, for *Length* bytes, is written starting at the current end of file.

If the call to ZwCreateFile set either of the *CreateOptions* flags, FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **ZwWriteFile** can specify that the current file position offset be used instead of an explicit *ByteOffset* value. This specification can be made by using one of the following methods:

- Specify a pointer to a LARGE_INTEGER value with the **HighPart** member set to -1 and the **LowPart** member set to the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a **NULL** pointer for *ByteOffset*.

ZwWriteFile updates the current file position by adding the number of bytes written when it completes the write operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **ZwWriteFile**. Doing this automatically changes the current file position to that *ByteOffset*value, performs the write operation, and then updates the position according to the number of bytes actually written. This technique gives the caller atomic seek-and-write service.

It is also possible to cause a write operation to start at the current end of file by specifying for *ByteOffset* a pointer to a LARGE_INTEGER value with **HighPart** set to -1 and **LowPart** set to FILE_WRITE_TO_END_OF_FILE. This works regardless of whether the I/O Manager is maintaining the current file position.

`Key`

Device and intermediate drivers should set this pointer to **NULL**.

Return Value

ZwWriteFile returns STATUS_SUCCESS on success or the appropriate NTSTATUS error code on failure.

Remarks

Callers of **ZwWriteFile** must have already called ZwCreateFile with the FILE_WRITE_DATA, FILE_APPEND_DATA, or GENERIC_WRITE flag set in the *DesiredAccess* parameter. Note that having only FILE_APPEND_DATA access to a file does not allow the caller to write anywhere in the file except at the current end-of-file mark, while having FILE_WRITE_DATA access to a file does not preclude the caller from writing to or beyond the end of a file.

If the preceding call to ZwCreateFile set the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING, the *Length* and *ByteOffset* parameters to **ZwWriteFile** must be an integral of the sector size. For more information, see **ZwCreateFile**.

ZwWriteFile begins the write operation to the file at *ByteOffset*, at the current file position, or at the end-of-file mark. It terminates the write operation when it has written *Length* bytes from *Buffer*. If necessary, it extends the length of the file and resets the end-of-file mark.

If the caller opened the file with the *DesiredAccess* SYNCHRONIZE flag set, the caller can wait for this routine to set the given *FileHandle* to the signaled state.

Drivers should call **ZwWriteFile** in the context of the system process in three cases:

1. The driver creates the file handle that it passes to **ZwWriteFile**.
2. **ZwWriteFile** notifies the driver of I/O completion by means of an event created by the driver.

3. **ZwWriteFile** notifies the driver of I/O completion by means of an APC callback routine that the driver passes to **ZwWriteFile**.

File and event handles are only valid in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **ZwWriteFile** in the context of the system process instead of the process context that the driver is in.

Likewise, **ZwWriteFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread issuing the I/O request. If the driver calls **ZwWriteFile** in the context of a process other than the system process, the APC could be delayed indefinitely, or it might not fire at all as the originating thread may never enter an alertable wait state.

For more information about working with files, see Using Files in a Driver.

Callers of **ZwWriteFile** must be running at IRQL = PASSIVE_LEVEL and with special kernel APCs enabled.

> **Note** If the call to this function occurs in user mode, you should use the name "NtWriteFile" instead of "**ZwWriteFile**".

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdm.h (include Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	PowerIrpDDis, HwStorPortProhibitedDDIs

See Also

KeInitializeEvent

ZwCreateFile

ZwQueryInformationFile

ZwReadFile

ZwSetInformationFile

wdmsec.h header

1/8/2019 • 2 minutes to read • Edit Online

This header header exposes security routines for kernel-mode drivers. They are used to create the device object with a security descriptor. Do not use the functions in this header directly, instead use:

- **IoCreateDeviceSecure**
- **IoValidateDeviceIoControlAccess**
- **RtlInitUnicodeString** wdmsec.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
WdmlibIoCreateDeviceSecure	The WdmlibIoCreateDeviceSecure function (or IoCreateDeviceSecure) creates a named device object and applies the specified security settings.
WdmlibIoValidateDeviceIoControlAccess	The WdmlibIoValidateDeviceIoControlAccess function verifies that the sender of an IRP_MJ_DEVICE_CONTROL or IRP_MJ_FILE_SYSTEM_CONTROL IRP has the specified access to the device object.
WdmlibRtlInitUnicodeStringEx	The WdmlibRtlInitUnicodeStringEx function initializes a counted string of Unicode characters.

WdmlibIoCreateDeviceSecure function

1/25/2019 • 4 minutes to read • Edit Online

The **WdmlibIoCreateDeviceSecure** function (or **IoCreateDeviceSecure**) creates a named device object and applies the specified security settings.

Syntax

```
NTSTATUS WdmlibIoCreateDeviceSecure(
  PDRIVER_OBJECT   DriverObject,
  ULONG            DeviceExtensionSize,
  PUNICODE_STRING  DeviceName,
  DEVICE_TYPE      DeviceType,
  ULONG            DeviceCharacteristics,
  BOOLEAN          Exclusive,
  PCUNICODE_STRING DefaultSDDLString,
  LPCGUID          DeviceClassGuid,
  PDEVICE_OBJECT   *DeviceObject
);
```

Parameters

`DriverObject`

Pointer to the driver object for the caller. Each driver receives a pointer to its driver object in a parameter to its DriverEntry routine. WDM function and filter drivers also receive a driver object pointer in their AddDevice routines.

`DeviceExtensionSize`

Specifies the driver-determined number of bytes to be allocated for the device extension of the device object. The internal structure of the device extension is driver-defined.

`DeviceName`

Optionally points to a buffer that contains a null-terminated Unicode string that names the device object. The string must be a full path name. If a name is not supplied, the FILE_AUTOGENERATED_DEVICE_NAME flag must be present in the *DeviceCharacteristics* parameter. (To create an unnamed device object, use the IoCreateDevice routine.)

`DeviceType`

Specifies one of the system-defined FILE_DEVICE_*XXX* constants that indicate the type of device (such as FILE_DEVICE_DISK, FILE_DEVICE_KEYBOARD, and so on), or a vendor-defined value for a new type of device. For more information, see Specifying Device Types. (Because a bus driver might not have information about a device's type, a device type value for a PDO can be specified in an INF AddReg directive.)

`DeviceCharacteristics`

Specifies one or more system-defined constants, ORed together, that provide additional information about the driver's device. For a list of possible device characteristics, see DEVICE_OBJECT. For more information about how to specify device characteristics, see Specifying Device Characteristics. Most drivers specify FILE_DEVICE_SECURE_OPEN for this parameter.

`Exclusive`

Specifies if the device object represents an *exclusive device*. Most drivers set this value to **FALSE**. For more information, see Specifying Exclusive Access to Device Objects.

`DefaultSDDLString`

Specifies a string representation for the default security settings of the device object. The security that is applied to the device object is derived from this string unless the system administrator places an override in the section of the registry identified by the *DeviceClassGuid* parameter.

The security setting is specified in a subset of Security Descriptor Definition Language (SDDL). A set of predefined constants (SDDL_DEVOBJ_*XXX*) are also provided. For more information, see Securing Device Objects.

`DeviceClassGuid`

Pointer to a GUID that identifies a section of the registry containing possible overrides for the *DefaultSDDLString*, *DeviceType*, *DeviceCharacteristics*, and *Exclusive* parameters.

> **Note** You should always specify a custom class GUID. You should not specify an existing class GUID. If you specify an existing class GUID, other drivers that attempt to specify that existing class GUID might fail to install or might install with incorrect security settings.

`DeviceObject`

Pointer to a variable that receives a pointer to the newly created DEVICE_OBJECT structure. The **DEVICE_OBJECT** structure is allocated from nonpaged pool.

Return Value

WdmlibIoCreateDeviceSecure returns STATUS_SUCCESS on success, or the appropriate NTSTATUS error code on failure. A partial list of the failure codes that could be returned by this function include:

Remarks

WdmlibIoCreateDeviceSecure creates a named device object, applies the specified security settings, and returns a pointer to the object. The caller is responsible for deleting the object when it is no longer needed by calling IoDeleteDevice.

This routine is not part of the operating system. Drivers can use the routine by linking to Wdmsec.lib on Microsoft Windows 2000 and later versions of Windows. (The Wdmsec.lib library first shipped with the Microsoft Windows XP Service Pack 1 [SP1] and Windows Server 2003 editions of the Driver Development Kit [DDK] and also ships with the Windows Driver Kit [WDK].)

Any driver that creates a named device object which is not guaranteed to have its security descriptor set by the INF file must use **WdmlibIoCreateDeviceSecure**. For more information, see Creating a Device Object. The caller is responsible for setting certain members of the returned device object. For more information, see Initializing a Device Object and the device-type-specific documentation for your device.

Be careful to specify the *DeviceType* and *DeviceCharacteristics* values in the correct parameters. Both parameters use system-defined FILE_*XXX* constants and some driver writers specify the values in the wrong parameters by mistake.

The caller is responsible for setting certain fields in the returned device object, such as the **Flags** field, and for initializing the device extension with any driver-defined information. For other operations that are required on new device objects, see Initializing a Device Object and the device-type-specific documentation for your device.

The GUID specified by the *DeviceClassGuid* parameter determines the device setup class for the device object. (For more information about device setup classes, see Device Setup Classes.) Callers must provide a value so that system administrators can change the security settings for the device (for instance, to deny access to certain users). For more information, see Setting Device Object Registry Properties After Installation.

Non-WDM drivers specify a new GUID, one not already in use by an existing device setup class. Generate a new GUID by using the GuidGen.exe tool that is included in the Microsoft Windows SDK.

WDM bus drivers that handle raw-mode capable devices can specify the device setup class of the device, but only if that class is guaranteed to already have been created. Otherwise, create a new GUID.

Device objects for disks, tapes, CD-ROMs, and RAM disks are given a Volume Parameter Block (VPB) that is initialized to indicate that the volume has never been mounted on the device.

If a driver's call to **WdmlibIoCreateDeviceSecure** returns an error, the driver should release any resources that it allocated for that device.

Requirements

Minimum supported client	See Remarks section.
Target Platform	Desktop
Header	wdmsec.h (include Wdmsec.h)
Library	Wdmsec.lib
IRQL	PASSIVE_LEVEL
DDI compliance rules	MiniportOnlyWdmDevice, HwStorPortProhibitedDDIs

See Also

DEVICE_OBJECT

IoAttachDevice

IoAttachDeviceToDeviceStack

IoCreateDevice

IoCreateSymbolicLink

IoDeleteDevice

WdmlibIoValidateDeviceIoControlAccess function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibIoValidateDeviceIoControlAccess** function verifies that the sender of an IRP_MJ_DEVICE_CONTROL or IRP_MJ_FILE_SYSTEM_CONTROL IRP has the specified access to the device object.

Syntax

```
NTSTATUS WdmlibIoValidateDeviceIoControlAccess(
  PIRP  Irp,
  ULONG RequiredAccess
);
```

Parameters

`Irp`

Specifies the IRP on which to perform the access check.

`RequiredAccess`

Specifies the type of access to the device object that the request sender must have. The caller can specify one or more of the following flags.

VALUE	MEANING
FILE_READ_ACCESS	The request sender must have read access to the device object
FILE_WRITE_ACCESS	The request sender must have write access to the device object.

Return Value

WdmlibIoValidateDeviceIoControlAccess returns an NTSTATUS value. Possible return values include:

RETURN CODE	DESCRIPTION

STATUS_SUCCESS	The request sender has the necessary access to the device object.
STATUS_ACCESS_DENIED	The request sender does not have the necessary access to the device object.
STATUS_INVALID_PARAMETER	The specified parameters are invalid. For example, if the routine is passed an IRP that is not an **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_FILE_SYSTEM_CONTROL** IRP, it returns STATUS_INVALID_PARAMETER.

Remarks

WdmlibIoValidateDeviceIoControlAccess allows drivers to perform dynamic access checks for IOCTLs. Use this routine to require more restrictive access than that specified in the IOCTL's definition. If the routine returns STATUS_ACCESS_DENIED, then the driver can complete the request with the STATUS_ACCESS_DENIED status value.

For example, if an IOCTL is defined with a *RequiredAccess* value of FILE_ANY_ACCESS, then by default any request sender with SYNCHRONIZE access to the device object can send the IOCTL. Use **WdmlibIoValidateDeviceIoControlAccess** to require more stringent security at run time. For more information about the *RequiredAccess* value of an IOCTL, see Defining I/O Control Codes.

The access checks are only performed if the **RequestorMode** member of the IRP structure is **UserMode**. If **RequestorMode** is **KernelMode**, the routine automatically returns STATUS_SUCCESS.

Requirements

Minimum supported client	Available in Windows Server 2003 and later versions of Windows. Drivers that must also work for Windows 2000 and Windows XP can instead link to Wdmsec.lib to use this routine. (The Wdmsec.lib library first shipped with the Windows XP Service Pack 1 [SP1] and Windows Server 2003 editions of the Driver Development Kit [DDK] and now ships with the Windows Driver Kit [WDK].)
Target Platform	Universal

Header	wdmsec.h (include Wdmsec.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level

See Also

IRP

WdmlibRtlInitUnicodeStringEx function

1/25/2019 • 2 minutes to read • Edit Online

The **WdmlibRtlInitUnicodeStringEx** function initializes a counted string of Unicode characters.

Syntax

```
NTSTATUS WdmlibRtlInitUnicodeStringEx(
  PUNICODE_STRING DestinationString,
  PCWSTR          SourceString
);
```

Parameters

`DestinationString`

A pointer to the UNICODE_STRING structure to be initialized.

`SourceString`

A pointer to a null-terminated wide-character string. This string is used to initialize the counted string pointed to by *DestinationString*.

Return Value

None

Remarks

The routine copies the *SourceString* pointer value to the **Buffer** member of the UNICODE_STRING structure pointed to by *DestinationString*. The **Length** member of this structure is set to the length, in bytes, of the source string, excluding the terminating null. The **MaximumLength** member of the structure is set to the length, in bytes, of the source string, including the terminating null. If *SourceString* is **NULL**, **Length** and **MaximumLength** are both set to zero.

WdmlibRtlInitUnicodeStringEx does not alter the source string pointed to by *SourceString*.

> **Note** If the source string is longer than MAX_USTRING - 1 bytes, **WdmlibRtlInitUnicodeStringEx** sets the **Length** member of the **UNICODE_STRING** structure pointed to by *DestinationString* to MAX_USTRING - 2, and sets the **MaximumLength** member of this structure to MAX_USTRING. In this case, the **Length** and **MaximumLength** values misrepresent the length of the null-terminated source string, and relying on the accuracy of these values is potentially dangerous.

Callers of **WdmlibRtlInitUnicodeStringEx** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL = PASSIVE_LEVEL because most other **Rtl*Xxx*String** routines cannot be called at IRQL > PASSIVE_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wdmsec.h (include Wdmsec.h, Wdm.h, Ntddk.h, Ntifs.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

RtlUnicodeStringInit

RtlUnicodeStringInitEx

UNICODE_STRING

wmidata.h header

1/8/2019 • 2 minutes to read • Edit Online

This header is used in providing kernel-mode Windows Management Instrumentation (WMI) extensions to WDM. For more information, see:

- Implementing WMI wmidata.h contains the following programming interfaces:

Structures

TITLE	DESCRIPTION
WMI_CHANGER_PROBLEM_DEVICE_ERROR	When the ChangerPerformDiagnostics routine performs diagnostic tests on a changer device it returns the results in a WMI_CHANGER_PROBLEM_DEVICE_ERROR structure.

wmilib.h header

1/8/2019 • 2 minutes to read • Edit Online

TThis header is used in providing kernel-mode Windows Management Instrumentation (WMI) extensions to WDM.

Drivers can use these routines in processing WMI IRPs.

- **WmiCompleteRequest**
- **WmiSystemControl**

To handle WMI IRPs by calling WmiSystemControl, a driver must implement certain required callback routines, for information, see Calling WmiSystemControl to Handle WMI IRPs

For more information, see:

- Implementing WMI wmilib.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
WmiCompleteRequest	The WmiCompleteRequest routine indicates that a driver has finished processing a WMI request in a DpWmiXxx routine.
WmiFireEvent	The WmiFireEvent routine sends an event to WMI for delivery to data consumers that have requested notification of the event.
WmiSystemControl	The WmiSystemControl routine is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.

Callback functions

TITLE	DESCRIPTION
WMI_EXECUTE_METHOD_CALLBACK	The DpWmiExecuteMethod routine executes a method associated with a data block. This routine is optional.
WMI_FUNCTION_CONTROL_CALLBACK	The DpWmiFunctionControl routine enables or disables notification of events, and enables or disables data collection for data blocks that the driver registered as expensive to collect. This routine is optional.
WMI_QUERY_DATABLOCK_CALLBACK	The DpWmiQueryDataBlock routine returns either a single instance or all instances of a data block. This routine is required.
WMI_QUERY_REGINFO_CALLBACK	The DpWmiQueryReginfo routine provides information about the data blocks and event blocks to be registered by a driver. This routine is required.

TITLE	DESCRIPTION
WMI_SET_DATABLOCK_CALLBACK	The DpWmiSetDataBlock routine changes all data items in a single instance of a data block. This routine is optional.
WMI_SET_DATAITEM_CALLBACK	The DpWmiSetDataItem routine changes a single data item in an instance of a data block. This routine is optional.

Structures

TITLE	DESCRIPTION
WMIGUIDREGINFO	The WMIGUIDREGINFO structure contains registration information for a given data block or event block exposed by a driver that uses the WMI library support routines.
WMILIB_CONTEXT	The WMILIB_CONTEXT structure provides registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

WMIGUIDREGINFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **WMIGUIDREGINFO** structure contains registration information for a given data block or event block exposed by a driver that uses the WMI library support routines.

Syntax

```
typedef struct _WMIGUIDREGINFO {
  LPCGUID Guid;
  ULONG   InstanceCount;
  ULONG   Flags;
} WMIGUIDREGINFO, *PWMIGUIDREGINFO;
```

Members

`Guid`

Pointer to the GUID that identifies the block. The memory that contains the GUID can be paged unless it is also used to call WmiFireEvent.

`InstanceCount`

Specifies the number of instances defined for the block.

`Flags`

Flag bits that indicate characteristics of the block. These flag bits are defined in the Wmistr.h header file. WMI ORs the **Flags** parameter value with the flag bits set by the driver in the *RegFlags* parameter of its DpWmiQueryReginfo routine, which apply to all of the data blocks and event blocks registered by the driver. **Flags** therefore supplements the driver's default settings for a given block.

A driver might set the following flag bit in **Flags**:

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, the *Pdo* parameter of the driver's *DpWmiQueryReginfo* routine points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flag bits:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens the data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIUPDATE).

Remarks

A driver that handles WMI IRPs by calling WMI library support routines builds an array of **WMIGUIDREGINFO** structures, one for each data block and event block to be registered. The driver sets the **GuidList** member of its WMILIB_CONTEXT structure to point to the first **WMIGUIDREGINFO** in the series.

Memory for this structure can be allocated from paged pool.

Requirements

Header	wmilib.h (include Wmilib.h)

See Also

DpWmiQueryReginfo

IRP_MN_DISABLE_COLLECTION

IRP_MN_ENABLE_COLLECTION

IRP_MN_REGINFO

IRP_MN_REGINFO_EX

WMILIB_CONTEXT

WmiFireEvent

WMILIB_CONTEXT structure

1/30/2019 • 2 minutes to read • Edit Online

The **WMILIB_CONTEXT** structure provides registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

Syntax

```
typedef struct _WMILIB_CONTEXT {
  ULONG                 GuidCount;
  PWMIGUIDREGINFO       GuidList;
  PWMI_QUERY_REGINFO    QueryWmiRegInfo;
  PWMI_QUERY_DATABLOCK  QueryWmiDataBlock;
  PWMI_SET_DATABLOCK    SetWmiDataBlock;
  PWMI_SET_DATAITEM     SetWmiDataItem;
  PWMI_EXECUTE_METHOD   ExecuteWmiMethod;
  PWMI_FUNCTION_CONTROL WmiFunctionControl;
} WMILIB_CONTEXT, *PWMILIB_CONTEXT;
```

Members

`GuidCount`

Specifies the number of blocks registered by the driver.

`GuidList`

Pointer to an array of **GuidCount** WMIGUIDREGINFO structures that contain registration information for each block.

`QueryWmiRegInfo`

Pointer to the driver's DpWmiQueryReginfo routine, which is a required entry point for drivers that call WMI library support routines.

`QueryWmiDataBlock`

Pointer to the driver's DpWmiQueryDataBlock routine, which is a required entry point for drivers that call WMI library support routines.

`SetWmiDataBlock`

Pointer to the driver's DpWmiSetDataBlock routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to **NULL**. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_INSTANCE request.

`SetWmiDataItem`

Pointer to the driver's DpWmiSetDataItem routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to **NULL**. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_ITEM request.

`ExecuteWmiMethod`

Pointer to the driver's DpWmiExecuteMethod routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to **NULL**. In this case, WMI returns STATUS_INVALID_DEVICE_REQUEST to the caller in response to any IRP_MN_EXECUTE_METHOD request.

`WmiFunctionControl`

Pointer to the driver's DpWmiFunctionControl routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to **NULL**. In this case, WMI returns STATUS_SUCCESS to the caller in response to any **IRP_MN_ENABLE_*XXX*** or **IRP_MN_DISABLE_*XXX*** request.

Remarks

A driver that handles WMI IRPs by calling WMI library support routines stores an initialized **WMILIB_CONTEXT** structure (or a pointer to such a structure) in its device extension. A driver can use the same **WMILIB_CONTEXT** structure for multiple device objects if each device object supplies the same set of data blocks.

When the driver receives an IRP_MJ_SYSTEM_CONTROL request, it calls WmiSystemControl with a pointer to its **WMILIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** determines whether the IRP contains a WMI request and, if so, handles the request by calling the driver's appropriate *DpWmiXxx* routine.

Memory for this structure can be allocated from paged pool.

Requirements

Header	wmilib.h (include Wmilib.h)

See Also

DpWmiExecuteMethod

DpWmiFunctionControl

DpWmiQueryDataBlock

DpWmiQueryReginfo

DpWmiSetDataBlock

DpWmiSetDataItem

WMIGUIDREGINFO

WmiSystemControl

WMI_EXECUTE_METHOD_CALLBACK callback function

1/25/2019 • 3 minutes to read • Edit Online

The *DpWmiExecuteMethod* routine executes a method associated with a data block. This routine is optional.

Syntax

```
WMI_EXECUTE_METHOD_CALLBACK WmiExecuteMethodCallback;

NTSTATUS WmiExecuteMethodCallback(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  ULONG GuidIndex,
  ULONG InstanceIndex,
  ULONG MethodId,
  ULONG InBufferSize,
  ULONG OutBufferSize,
  PUCHAR Buffer
)
{...}
```

Parameters

`DeviceObject`

Pointer to the driver's WDM DEVICE_OBJECT structure.

`Irp`

Pointer to the IRP.

`GuidIndex`

Specifies the data block by supplying a zero-based index into the list of GUIDs that the driver provided in the WMILIB_CONTEXT structure it passed to WmiSystemControl.

`InstanceIndex`

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies a zero-based index value that identifies the instance.

`MethodId`

Specifies the ID of the method to execute. The driver defines the method ID as an item in a data block.

`InBufferSize`

Indicates the size in bytes of the input data. If there is no input data, *InBufferSize* is zero.

`OutBufferSize`

Indicates the number of bytes available in the buffer for output data.

`Buffer`

Pointer to a buffer that holds input data, if any, and receives output data, if any, for the method. If the buffer is too small to receive all of the output data, the driver returns STATUS_BUFFER_TOO_SMALL and calls WmiCompleteRequest with the size required.

Return Value

DpWmiExecuteMethod returns STATUS_SUCCESS or an appropriate error code such as the following:

Remarks

WMI calls a driver's *DpWmiExecuteMethod* routine after the driver calls WmiSystemControl in response to an IRP_MN_EXECUTE_METHOD request.

If a driver implements a *DpWmiExecuteMethod* routine, the driver must place the routine's address in the **ExecuteWmiMethod** member of the WMILIB_CONTEXT structure that it passes to WmiSystemControl. If a driver does not implement a *DpWmiExecuteMethod* routine, it must set **ExecuteWmiMethod** to **NULL**. In the latter case, WMI returns STATUS_INVALID_DEVICE_REQUEST to the caller in response to any **IRP_MN_EXECUTE_METHOD** request.

The driver is responsible for validating all input arguments. Specifically, the driver must do the following:

- Verify that the *GuidIndex* value is between zero and GuidCount-1, based on the **GuidCount** member of the WMILIB_CONTEXT structure.
- Verify that the driver has not flagged the specified data block for removal. If the driver recently specified the WMIREG_FLAG_REMOVE_GUID flag in a WMIGUIDREGINFO structure that is contained in a **WMILIB_CONTEXT** structure, it is possible for an additional request to arrive before the removal occurs.
- Verify that the *InstanceIndex* value is within the range of instance indexes that are supported by the driver for the data block.
- Verify that the method identifier supplied by *MethodId* is a valid identifier for the specified data block, and that the caller is allowed to execute the method.
- Verify that *Buffer* and *InBufferSize* describe a buffer that is large enough to contain the specified method's input parameters, including padding if necessary, and verify that the input parameters are valid.
- Verify that *Buffer* and *OutBufferSize* describe a buffer that is large enough to receive the specified method's output data, including padding if necessary.

Do not assume the thread context is that of the initiating user-mode application—a higher-level driver might have changed it.

If the specified method performs an operation that causes data loss, such as fetching and resetting the contents of a set of counters, the driver should validate the output buffer size before performing the operation. That way, the driver can return STATUS_BUFFER_TOO_SMALL and allow the caller to resubmit the request with a larger buffer, without prematurely resetting the counters.

After executing the method and writing any output data to the buffer, the driver calls WmiCompleteRequest to complete the request.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_EXECUTE_METHOD

WMILIB_CONTEXT

WmiCompleteRequest

WmiSystemControl

WMI_FUNCTION_CONTROL_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DpWmiFunctionControl* routine enables or disables notification of events, and enables or disables data collection for data blocks that the driver registered as expensive to collect. This routine is optional.

Syntax

```
WMI_FUNCTION_CONTROL_CALLBACK WmiFunctionControlCallback;

NTSTATUS WmiFunctionControlCallback(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  ULONG GuidIndex,
  WMIENABLEDISABLECONTROL Function,
  BOOLEAN Enable
)
{...}
```

Parameters

`DeviceObject`

Pointer to the driver's WDM DEVICE_OBJECT structure.

`Irp`

Pointer to the IRP.

`GuidIndex`

Specifies the data block by supplying a zero-based index into the list of GUIDs that the driver provided in the WMILIB_CONTEXT structure it passed to WmiSystemControl.

`Function`

Specifies what to enable or disable. **WmiEventControl** indicates an event, and **WmiDataBlockControl** indicates data collection for a block that was registered as expensive to collect (that is, a block for which the driver set WMIREG_FLAG_EXPENSIVE in **Flags** of the WMIGUIDREGINFO structure used to register the block).

`Enable`

Specifies **TRUE** to enable the event or data collection, or **FALSE** to disable it.

Return Value

DpWmiFunctionControl returns STATUS_SUCCESS or an appropriate error status such as:

Remarks

WMI calls a driver's *DpWmiFunctionControl* routine after the driver calls WmiSystemControl in response to one of the following requests:

IRP_MN_ENABLE_COLLECTION

IRP_MN_DISABLE_COLLECTION

IRP_MN_ENABLE_EVENTS

IRP_MN_DISABLE_EVENTS

If a driver implements a *DpWmiFunctionControl* routine, the driver must place the routine's address in the **WmiFunctionControl** member of the WMILIB_CONTEXT structure that it passes to WmiSystemControl. If a driver does not implement a *DpWmiFunctionControl* routine, it must set **WmiFunctionControl** to **NULL**. In the latter case, WMI returns STATUS_SUCCESS to the caller.

The driver is responsible for validating all input arguments. Specifically, the driver must do the following:

- Verify that the *GuidIndex* value is between zero and GuidCount-1, based on the **GuidCount** member of the WMILIB_CONTEXT structure.
- Verify that the driver has not flagged the specified data block for removal. If the driver recently specified the WMIREG_FLAG_REMOVE_GUID flag in a WMIGUIDREGINFO structure that is contained in a **WMILIB_CONTEXT** structure, it is possible for an additional request to arrive before the removal occurs.

It is unnecessary for the driver to check whether events or data collection are already enabled for a block, because WMI sends a single enable request when the first data consumer enables the block, and it sends a single disable request when the last data consumer disables the block. WMI will not call *DpWmiFunctionControl* to enable a block without an intervening call to disable it.

After enabling or disabling the event or data collection for the block, the driver calls WmiCompleteRequest to complete the request.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_DISABLE_COLLECTION

IRP_MN_DISABLE_EVENTS

IRP_MN_ENABLE_COLLECTION

IRP_MN_ENABLE_EVENTS

WMILIB_CONTEXT

WmiSystemControl

WMI_QUERY_DATABLOCK_CALLBACK callback function

1/25/2019 • 3 minutes to read • Edit Online

The *DpWmiQueryDataBlock* routine returns either a single instance or all instances of a data block. This routine is required.

Syntax

```
WMI_QUERY_DATABLOCK_CALLBACK WmiQueryDatablockCallback;

NTSTATUS WmiQueryDatablockCallback(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  ULONG GuidIndex,
  ULONG InstanceIndex,
  ULONG InstanceCount,
  PULONG InstanceLengthArray,
  ULONG BufferAvail,
  PUCHAR Buffer
)
{...}
```

Parameters

`DeviceObject`

Pointer to the driver's WDM DEVICE_OBJECT structure.

`Irp`

Pointer to the IRP.

`GuidIndex`

Specifies the data block by supplying a zero-based index into the list of GUIDs that the driver provided in the WMILIB_CONTEXT structure it passed to WmiSystemControl.

`InstanceIndex`

If *DpWmiQueryDataBlock* is called in response to an IRP_MN_QUERY_SINGLE_INSTANCE request, *InstanceIndex* specifies a zero-based index that indicates the instance of the specified data block to be queried. If *DpWmiQueryDataBlock* is called in response to an IRP_MN_QUERY_ALL_DATA request, *InstanceIndex*is zero.

`InstanceCount`

If *DpWmiQueryDataBlock* is called in response to an IRP_MN_QUERY_SINGLE_INSTANCE request, *InstanceCount*is 1. If *DpWmiQueryDataBlock* is called in response to an IRP_MN_QUERY_ALL_DATA request, *InstanceCount*is the number of instances to be returned.

`InstanceLengthArray`

Pointer to a caller-supplied, *InstanceCount*-sized array of ULONG elements. The driver fills in each array element to indicate the length of each instance that was returned. If *BufferAvail* is zero, *InstanceLengthArray* is **NULL**.

`BufferAvail`

Specifies the maximum number of bytes that are available to receive data in the buffer at *Buffer*. If this value is zero, the caller is requesting that the driver specify the required buffer size in its call to WmiCompleteRequest. See the Remarks section for more information.

`Buffer`

Pointer to the buffer to receive instance data. If the buffer is large enough to receive all of the data, the driver writes the instance data to the buffer with each instance aligned on an 8-byte boundary. If the buffer is too small to receive all of the data, the driver calls WmiCompleteRequest with *BufferUsed* set to the size required.

Return Value

DpWmiQueryDataBlock returns STATUS_SUCCESS or an error status such as the following:

If the driver cannot complete the request immediately, it can return STATUS_PENDING.

Remarks

WMI calls a driver's *DpWmiQueryDataBlock* routine after the driver calls WmiSystemControl in response to an IRP_MN_QUERY_SINGLE_INSTANCE or IRP_MN_QUERY_ALL_DATA request. The driver must place the address of its *DpWmiQueryDataBlock* routine in the WMILIB_CONTEXT structure that it passes to **WmiSystemControl**.

The driver is responsible for validating all input arguments. Specifically, the driver must do the following:

- Verify that the *GuidIndex* value is between zero and GuidCount-1, based on the **GuidCount** member of the WMILIB_CONTEXT structure.
- Verify that the driver has not flagged the specified data block for removal. If the driver recently specified the WMIREG_FLAG_REMOVE_GUID flag in a WMIGUIDREGINFO structure that is contained in a **WMILIB_CONTEXT** structure, it is possible for a query to arrive before the removal occurs.
- Verify that the *InstanceIndex* and *InstanceCount* values, together, specify a number of data block instances that is within the range supported by the driver for the data block. If *InstanceIndex* is nonzero, *InstanceCount* must be 1. If *InstanceIndex* is 0, *InstanceCount* must not be greater than the number of instances supported.
- Verify that the buffer described by *Buffer* and *BufferAvail* is large enough to hold all requested instances of the data block. Each instance must begin on an 8-byte boundary, and additional padding might exist between data items within each instance.

After writing instance data to the buffer, the driver calls WmiCompleteRequest to complete the request. If the buffer described by *Buffer* and *BufferAvail* is zero, or is too small to receive all the requested data, the call to **WmiCompleteRequest** must specify STATUS_BUFFER_TOO_SMALL for the *Status* parameter and the required buffer size for the *BufferUsed* parameter.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)

IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_QUERY_ALL_DATA

IRP_MN_QUERY_SINGLE_INSTANCE

WMILIB_CONTEXT

WmiCompleteRequest

WmiSystemControl

WMI_QUERY_REGINFO_CALLBACK callback function

1/25/2019 • 4 minutes to read • Edit Online

The *DpWmiQueryReginfo* routine provides information about the data blocks and event blocks to be registered by a driver. This routine is required.

Syntax

```
WMI_QUERY_REGINFO_CALLBACK WmiQueryReginfoCallback;

NTSTATUS WmiQueryReginfoCallback(
  PDEVICE_OBJECT DeviceObject,
  PULONG RegFlags,
  PUNICODE_STRING InstanceName,
  PUNICODE_STRING *RegistryPath,
  PUNICODE_STRING MofResourceName,
  PDEVICE_OBJECT *Pdo
)
{...}
```

Parameters

`DeviceObject`

A pointer to the driver's WDM DEVICE_OBJECT structure.

`RegFlags`

This parameter indicates common characteristics of all blocks being registered. Any flag set in *RegFlags* is applied to all blocks. A driver can supplement *RegFlags* for a given block by setting **Flags** in the block's WMIGUIDREGINFO structure. For example, a driver might clear WMIREG_FLAG_EXPENSIVE in *RegFlags*, but set it in **Flags** to register a given block as expensive to collect.

The driver sets one of the following flags in *RegFlags*:

WMIREG_FLAG_INSTANCE_BASENAME

Requests WMI to generate static instance names from a base name provided by the driver at the *InstanceName*. WMI generates instance names by appending a counter to the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If the driver sets this flag, it must also set *Pdo* to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flags in *RegFlags*, but more typically would set them in **Flags** of a block's WMIGUIDREGINFO structure:

WMIREG_FLAG_EVENT_ONLY_GUID

The blocks can be enabled or disabled as events only, and cannot be queried or set. If this flag is clear, the blocks can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens a data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for the blocks. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIUPDATE).

`InstanceName`

A pointer to a single counted Unicode string that serves as the base name for all instances of all blocks to be registered by the driver. WMI frees the string with ExFreePool. If WMIREG_FLAG_INSTANCE_BASENAME is clear, *InstanceName* is ignored.

`*RegistryPath`

A pointer to a pointer to a counted Unicode string that specifies the registry path passed to the driver's DriverEntry routine.

`MofResourceName`

A pointer to a single counted Unicode string that indicates the name of the MOF resource attached to the driver's binary image file. Typically this string would be a static defined by the driver. WMI makes a copy of this string after the driver returns from this routine. This string can be dynamically allocated by the driver. In the case of an allocated string, the driver is responsible for freeing the string which should be done after WmiSystemControl returns. If the driver does not have a MOF resource attached, it can leave *MofResourceName* unchanged.

`*Pdo`

A pointer to the physical device object (PDO) passed to the driver's AddDevice routine. If WMIREG_FLAG_INSTANCE_PDO is set, WMI uses the device instance path of this PDO as a base from which to generate static instance names. If WMIREG_FLAG_INSTANCE_PDO is clear, WMI ignores *Pdo*.

Return Value

DpWmiQueryReginfo always returns STATUS_SUCCESS.

Remarks

WMI calls a driver's *DpWmiQueryReginfo* routine after the driver calls WmiSystemControl in response to an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request. The driver must place the address of its *DpWmiQueryReginfo* routine in the WMILIB_CONTEXT structure that it passes to **WmiSystemControl**.

WMI sends the IRP after a driver calls IoWMIRegistrationControl with WMIREG_ACTION_REGISTER, WMIREG_ACTION_REREGISTER, or WMIREG_ACTION_UPDATE. WMI transparently handles the differences between **IRP_MN_REGINFO** and **IRP_MN_REGINFO_EX** on behalf of the driver.

WMI does not send an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_DEREGISTER, because WMI requires no further information from the driver. A driver typically deregisters its blocks in response to an IRP_MN_REMOVE_DEVICE request.

The driver provides new or updated registration information about individual blocks, or indicates blocks to remove, in the WMILIB_CONTEXT structure it passes to **WmiSystemControl**. After the initial call, which establishes the driver's registry path and MOF resource name, a driver's *DpWmiQueryReginfo* routine can change flags common to all of a driver's blocks, provide a different base name string used to generate instance names, or change the basis

for instance names from a string to the device instance path of the PDO.

The driver must not return STATUS_PENDING or block the request. The driver must not complete the request by calling WmiCompleteRequest from its *DpWmiQueryReginfo* routine or by calling IoCompleteRequest after **WmiSystemControl** returns.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_REGINFO_EX

IoWMIRegistrationControl

WMIGUIDREGINFO

WMILIB_CONTEXT

WmiSystemControl

WMI_SET_DATABLOCK_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DpWmiSetDataBlock* routine changes all data items in a single instance of a data block. This routine is optional.

Syntax

```
WMI_SET_DATABLOCK_CALLBACK WmiSetDatablockCallback;

NTSTATUS WmiSetDatablockCallback(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  ULONG GuidIndex,
  ULONG InstanceIndex,
  ULONG BufferSize,
  PUCHAR Buffer
)
{...}
```

Parameters

`DeviceObject`

Pointer to the driver's WDM DEVICE_OBJECT structure.

`Irp`

Pointer to the IRP.

`GuidIndex`

Specifies the data block by its zero-based index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to WmiSystemControl.

`InstanceIndex`

If the block specified by *GuidIndex*has multiple instances, *InstanceIndex* specifies the instance.

`BufferSize`

Specifies the size in bytes of the buffer at *Buffer*.

`Buffer`

Pointer to a buffer that contains new values for the instance.

Return Value

DpWmiSetDataBlock returns STATUS_SUCCESS or an appropriate error status such as the following:

If the driver cannot complete the request immediately, it can return STATUS_PENDING.

Remarks

WMI calls a driver's *DpWmiSetDataBlock* routine after the driver calls WmiSystemControl in response to an

IRP_MN_CHANGE_SINGLE_INSTANCE request.

The driver is responsible for validating all input arguments. Specifically, the driver must do the following:

- Verify that the *GuidIndex* value is between zero and GuidCount-1, based on the **GuidCount** member of the WMILIB_CONTEXT structure.
- Verify that the driver has not flagged the specified data block for removal. If the driver recently specified the WMIREG_FLAG_REMOVE_GUID flag in a WMIGUIDREGINFO structure that is contained in a **WMILIB_CONTEXT** structure, it is possible for a set request to arrive before the removal occurs.
- Verify that the *InstanceIndex* value is within the range of instance indexes that are supported by the driver for the data block.
- Verify that *Buffer* and *BufferSize* describe a valid-sized data block, including any padding that exists between data items, and that the contents of the buffer are valid for the data block.
- Verify that the specified data block is one for which the driver allows caller-initiated modifications. In other words, the driver should not allow modifications to data blocks that you intended to be read-only.

Do not assume the thread context is that of the initiating user-mode application—a higher-level driver might have changed it.

If a driver implements a *DpWmiSetDataBlock* routine, the driver must place the routine's address in the **SetWmiDataBlock** member of the WMILIB_CONTEXT structure that it passes to WmiSystemControl. If a driver does not implement a *DpWmiSetDataBlock* routine, it must set **SetWmiDataBlock** to **NULL**. In the latter case, WMI returns STATUS_READ_ONLY to the caller.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_CHANGE_SINGLE_INSTANCE

WMILIB_CONTEXT

WmiSystemControl

WMI_SET_DATAITEM_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DpWmiSetDataItem* routine changes a single data item in an instance of a data block. This routine is optional.

Syntax

```
WMI_SET_DATAITEM_CALLBACK WmiSetDataitemCallback;

NTSTATUS WmiSetDataitemCallback(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  ULONG GuidIndex,
  ULONG InstanceIndex,
  ULONG DataItemId,
  ULONG BufferSize,
  PUCHAR Buffer
)
{...}
```

Parameters

`DeviceObject`

Pointer to the driver's WDM DEVICE_OBJECT structure.

`Irp`

Pointer to the IRP.

`GuidIndex`

Specifies the data block by supplying a zero-based index into the list of GUIDs that the driver provided in the WMILIB_CONTEXT structure it passed to WmiSystemControl.

`InstanceIndex`

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* is a zero-based value that specifies the instance.

`DataItemId`

Specifies the ID of the data item to set.

`BufferSize`

Specifies the size in bytes of the buffer at *Buffer*.

`Buffer`

Pointer to a buffer that contains the new value for the data item.

Return Value

DpWmiSetDataItem returns STATUS_SUCCESS or an appropriate error code such as the following:

Remarks

WMI calls a driver's *DpWmiSetDataItem* routine after the driver calls WmiSystemControl in response to an IRP_MN_CHANGE_SINGLE_ITEM request.

Do not implement *DpWmiSetDataItem* unless you are sure that a system-supplied user-mode component requires this capability. If you implement a *DpWmiSetDataItem* routine, the driver must place the routine's address in the **SetWmiDataItem** member of the WMILIB_CONTEXT structure that it passes to WmiSystemControl. If you do not implement a *DpWmiSetDataItem* routine, the driver must set **SetWmiDataItem** to **NULL**. In the latter case, WMI returns STATUS_READ_ONLY to the caller.

The driver is responsible for validating all input arguments. Specifically, the driver must do the following:

- Verify that the *GuidIndex* value is between zero and GuidCount-1, based on the **GuidCount** member of the WMILIB_CONTEXT structure.
- Verify that the driver has not flagged the specified data block for removal. If the driver recently specified the WMIREG_FLAG_REMOVE_GUID flag in a WMIGUIDREGINFO structure that is contained in a **WMILIB_CONTEXT** structure, it is possible for a set request to arrive before the removal occurs.
- Verify that the *InstanceIndex* value is within the range of instance indexes that are supported by the driver for the data block.
- Verify that the *DataItemId* value is within the range of data item identifiers that are supported by the driver for the data block.
- Verify that *Buffer* and *BufferSize* describe a valid-sized data item, and that the contents of the buffer are valid for the data item.
- Verify that the specified data item is one for which the driver allows caller-initiated modifications. In other words, the driver should not allow modifications to data items that you intended to be read-only.

Do not assume the thread context is that of the initiating user-mode application—a higher-level driver might have changed it.

This routine can be pageable.

For more information about implementing this routine, see Calling WmiSystemControl to Handle WMI IRPs.

Requirements

Target Platform	Desktop
Header	wmilib.h (include Wmilib.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

IRP_MN_CHANGE_SINGLE_ITEM

WMILIB_CONTEXT

WmiSystemControl

WmiCompleteRequest function

1/25/2019 • 2 minutes to read • Edit Online

The **WmiCompleteRequest** routine indicates that a driver has finished processing a WMI request in a *DpWmiXxx* routine.

Syntax

```
NTSTATUS WmiCompleteRequest(
  PDEVICE_OBJECT DeviceObject,
  PIRP           Irp,
  NTSTATUS       Status,
  ULONG          BufferUsed,
  CCHAR          PriorityBoost
);
```

Parameters

`DeviceObject`

A pointer to the driver's DEVICE_OBJECT.

`Irp`

A pointer to the IRP.

`Status`

Specifies the status to return for the IRP.

`BufferUsed`

Specifies the number of bytes needed in the buffer passed to the driver's *DpWmiXxx* routine. If the buffer is too small, the driver sets *Status* to STATUS_BUFFER_TOO_SMALL and sets *BufferUsed* to the number of bytes needed for the data to be returned. If the buffer passed is large enough, the driver sets *BufferUsed* to the number of bytes actually used.

`PriorityBoost`

Specifies a system-defined constant by which to increment the run-time priority of the original thread that requested the operation. WMI calls IoCompleteRequest with *PriorityBoost* when it completes the IRP. See **IoCompleteRequest** for more information on *PriorityBoost*.

Return Value

WmiCompleteRequest returns the value that was passed to it in the *Status* parameter unless *Status* was set to STATUS_BUFFER_TOO_SMALL. If the driver set *Status* equal to STATUS_BUFFER_TOO_SMALL, **WmiCompleteRequest** builds a WNODE_TOO_SMALL structure and returns STATUS_SUCCESS. The return value from **WmiCompleteRequest** should be returned by the driver in its *DpWmiXxx* routine.

Remarks

A driver calls **WmiCompleteRequest** from a *DpWmiXxx* routine after it finishes all other processing in that

routine, or after the driver finishes all processing for a pending IRP. **WmiCompleteRequest** fills in a **WNODE_*XXX*** with any data returned by the driver and calls **IoCompleteRequest** to complete the IRP.

A driver should always return the return value from **WmiCompleteRequest** in its *DpWmiXxx* routine.

A driver must not call **WmiCompleteRequest** from its DpWmiQueryRegInfo routine.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wmilib.h (include Wmilib.h)
Library	Wmilib.lib
IRQL	"<= DISPATCH_LEVEL"

See Also

DpWmiExecuteMethod

DpWmiFunctionControl

DpWmiQueryDataBlock

DpWmiQueryReginfo

DpWmiSetDataBlock

DpWmiSetDataItem

IoCompleteRequest

WmiSystemControl

WmiFireEvent function

1/25/2019 • 2 minutes to read • Edit Online

The **WmiFireEvent** routine sends an event to WMI for delivery to data consumers that have requested notification of the event.

Syntax

```
NTSTATUS WmiFireEvent(
  PDEVICE_OBJECT DeviceObject,
  LPCGUID        Guid,
  ULONG          InstanceIndex,
  ULONG          EventDataSize,
  PVOID          EventData
);
```

Parameters

`DeviceObject`

A pointer to the driver's DEVICE_OBJECT.

`Guid`

A pointer to the GUID that represents the event block.

`InstanceIndex`

If the event block has multiple instances, specifies the index of the instance.

`EventDataSize`

Specifies the number of bytes of data at *EventData*. If no data is generated for an event, *EventData* must be zero.

`EventData`

A pointer to a driver-allocated nonpaged buffer containing data generated by the driver for the event. If no data is generated for an event, *EventData* must be **NULL**. WMI frees the buffer without further intervention by the driver.

Return Value

WmiFireEvent propagates the status returned by IoWmiWriteEvent, or returns STATUS_INSUFFICIENT_RESOURCES if it could not allocate memory for the event.

Remarks

A driver calls **WmiFireEvent** to send an event to WMI for delivery to all data consumers that have requested notification of the event. All pointers passed to **WmiFireEvent** must point to nonpagable memory, such as nonpaged pool.

The driver sends an event only if it has been previously enabled by the driver's DpWmiFunctionControl routine, which WMI calls to process an **IRP_MN_ENABLE_EVENT** request.

The driver writes any data associated with the event to the buffer at *EventData*. WMI fills in a

WNODE_SINGLE_INSTANCE structure with the data and calls IoWmiWriteEvent to deliver the event.

For more information about event tracing, see WMI Event Tracing.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wmilib.h (include Wmilib.h)
Library	Wmilib.lib
IRQL	"<= DISPATCH_LEVEL"

See Also

DpWmiFunctionControl

IRP_MN_ENABLE_EVENTS

WmiSystemControl

WmiSystemControl function

1/25/2019 • 2 minutes to read • Edit Online

The **WmiSystemControl** routine is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.

Syntax

```
NTSTATUS WmiSystemControl(
  PWMILIB_CONTEXT         WmiLibInfo,
  PDEVICE_OBJECT          DeviceObject,
  PIRP                    Irp,
  PSYSCTL_IRP_DISPOSITION IrpDisposition
);
```

Parameters

`WmiLibInfo`

A pointer to a WMILIB_CONTEXT structure that contains registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

`DeviceObject`

A pointer to the driver's DEVICE_OBJECT.

`Irp`

A pointer to the IRP.

`IrpDisposition`

A pointer to an enumeration value of type **SYSCTL_IRP_DISPOSITION** that indicates how the IRP was handled. **WmiSystemControl** always sets this value, even when it returns a non-success NTSTATUS code.

SYSCTL_IRP_DISPOSITION is an enumeration in Wmilib.h and contains the following values:

IrpProcessed

The IRP was processed and possibly completed. If the driver's *DpWmi*Xxx routine called by **WmiSystemControl** did not complete the IRP, the driver must call WmiCompleteRequest to complete the IRP after **WmiSystemControl** returns.

IrpNotCompleted

The IRP was processed but not completed, either because WMI detected an error and set up the IRP with an appropriate error code, or processed an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request. The driver must complete the IRP by calling IoCompleteRequest.

IrpNotWmi

The IRP is not a WMI request (that is, WMI does not recognize the IRP's minor code). If the driver handles IRP_MJ_SYSTEM_CONTROL requests with this **IRP_MN_*XXX***, it should handle the IRP; otherwise, the driver should forward the IRP to the next lower driver. If the driver is the lowest-level driver, then it must complete the IRP.

IrpForward

The IRP is targeted to another device object (that is, the device object pointer at **Parameters.WMI.ProviderId** in the IRP does not match the pointer passed by the driver in its call to IoWMIRegistrationControl). The driver must forward the IRP to the next lower driver. If the driver is the lowest-level driver, then it must complete the IRP.

Return Value

WmiSystemControl returns STATUS_SUCCESS or one of the following error codes:

Remarks

When a driver receives an **IRP_MJ_SYSTEM_CONTROL** request with a WMI IRP minor code, it calls **WmiSystemControl** with a pointer to the driver's **WMILIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. The **WMILIB_CONTEXT** structure contains registration information for the driver's data blocks and event blocks and defines entry points for its WMI library callback routines.

WmiSystemControl confirms that the IRP is a WMI request and determines whether the block specified by the request is valid for the driver. If so, it processes the IRP by calling the appropriate *DpWmi*Xxx entry point in the driver's **WMILIB_CONTEXT** structure. WMI is running at IRQL PASSIVE_LEVEL when it calls the driver's *DpWmi*Xxx routine.

A driver must be running at IRQL PASSIVE_LEVEL when it forwards an **IRP_MJ_SYSTEM_CONTROL** request to the next-lower driver.

Requirements

Minimum supported client	Available in Windows 2000 and later versions of Windows.
Target Platform	Universal
Header	wmilib.h (include Wmilib.h)
Library	Wmilib.lib
IRQL	PASSIVE_LEVEL (see Remarks section)
DDI compliance rules	WmiComplete

See Also

DpWmiExecuteMethod

DpWmiFunctionControl

DpWmiQueryDataBlock

DpWmiQueryReginfo

DpWmiSetDataBlock

DpWmiSetDataItem

IRP_MJ_SYSTEM_CONTROL

IRP_MN_REGINFO_EX

IoCompleteRequest

WMILIB_CONTEXT

wmistr.h header

1/8/2019 • 2 minutes to read • Edit Online

This header declares a set of standard data structures called WNODE_XXX that WMI uses to pass data between user-mode data consumers and kernel-mode data providers such as drivers.

This header is used by kernel. For more information, see:

- WMI WNODE_XXX Structures wmistr.h contains the following programming interfaces:

Structures

TITLE	DESCRIPTION
WMIREGGUIDW	The WMIREGGUID structure contains new or updated registration information for a data block or event block.
WMIREGINFOW	The WMIREGINFO structure contains information provided by a driver to register or update its data blocks and event blocks.
WNODE_ALL_DATA	The WNODE_ALL_DATA structure contains data for all instances of a data block or event block.
WNODE_EVENT_ITEM	The WNODE_EVENT_ITEM structure contains data generated by a driver for an event.
WNODE_EVENT_REFERENCE	The WNODE_EVENT_REFERENCE structure contains information that WMI can use to query for an event that exceeds the event size limit set in the registry.
WNODE_HEADER	The WNODE_HEADER structure is the first member of all other WNODE_XXX structures. It contains information common to all such structures.
WNODE_METHOD_ITEM	The WNODE_METHOD_ITEM structure indicates a method associated with an instance of a data block and contains any input data for the method.
WNODE_SINGLE_INSTANCE	The WNODE_SINGLE_INSTANCE structure contains values for all data items in one instance of a data block.
WNODE_SINGLE_ITEM	The WNODE_SINGLE_ITEM structure contains the value of a single data item in an instance of a data block.
WNODE_TOO_SMALL	The WNODE_TOO_SMALL structure indicates the size of the buffer needed to receive output from a request.

WMIREGGUIDW structure

1/25/2019 • 4 minutes to read • Edit Online

The **WMIREGGUID** structure contains new or updated registration information for a data block or event block.

Syntax

```
typedef struct {
  GUID  Guid;
  ULONG Flags;
  ULONG InstanceCount;
  union {
    ULONG     InstanceNameList;
    ULONG     BaseNameOffset;
    ULONG_PTR Pdo;
    ULONG_PTR InstanceInfo;
  } DUMMYUNIONNAME;
} WMIREGGUIDW, *PWMIREGGUIDW;
```

Members

`Guid`

Specifies the GUID that represents the block to register or update.

`Flags`

Indicates characteristics of the block to register or update.

If a block is being registered with static instance names, a driver sets one of the following flags:

WMIREG_FLAG_INSTANCE_LIST

Indicates that the driver provides static instance names for this block in a static list following the WMIREGINFO structure in the buffer at **IrpStack->Parameters.WMI.Buffer**. If this flag is set, **InstanceNameList** is the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a contiguous series of **InstanceCount** counted Unicode strings.

WMIREG_FLAG_INSTANCE_BASENAME

Requests WMI to generate static instance names from a base name provided by the driver following the **WMIREGINFO** structure in the buffer at **IrpStack->Parameters.WMI.Buffer**. WMI generates instance names by appending a counter to the base name. If this flag is set, **BaseNameOffset** is the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a single counted Unicode string that serves as the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, **InstanceInfo** points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

If a block is being registered with dynamic instance names, WMIREG_FLAG_INSTANCE_LIST, WMIREG_FLAG_INSTANCE_BASENAME, and WMIREG_FLAG_INSTANCE_PDO must be clear.

A driver might also set one or more of the following flags:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens the data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIUPDATE).

WMIREG_FLAG_TRACED_GUID

The block can be written only to a log file and can be accessed only through user-mode routines declared in evntrace.h. Only NT kernel-mode data providers set this flag.

WMIREG_FLAG_TRACE_CONTROL_GUID

The GUID acts as the control GUID for enabling or disabling the trace GUIDs associated with it in the MOF file. This flag is valid only if WMIREG_FLAG_TRACED_GUID is also set. Only NT kernel-mode data providers set this flag.

`InstanceCount`

Specifies the number of static instance names to be defined for this block. If the block is being registered with dynamic instance names, WMI ignores **InstanceCount**.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.InstanceNameList`

Indicates the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a contiguous series of **InstanceCount** counted Unicode strings. This member is valid only if WMIREG_FLAG_INSTANCE_LIST is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **InstanceNameList**.

`DUMMYUNIONNAME.BaseNameOffset`

Indicates the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a single counted Unicode string that serves as a base for WMI to generate static instance names. This member is valid only if WMIREG_FLAG_INSTANCE_BASENAME is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **BaseNameOffset**.

`DUMMYUNIONNAME.Pdo`

Pointer to the physical device object (PDO) passed to the driver's AddDevice routine. WMI uses the device instance path of this PDO as a base from which to generate static instance names. This member is valid only if WMIREG_FLAG_INSTANCE_PDO is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **Pdo**.

`DUMMYUNIONNAME.InstanceInfo`

Reserved for use by WMI.

Remarks

A driver builds one or more **WMIREGGUID** structures in response to an IRP_MN_REGINFO or IRP_MN_REGINFO_EX request to register or update its blocks. The driver passes an array of such structures at the **WmiRegGuid** member of a **WMIREGINFO** structure, which the driver writes to the buffer at **IrpStack->Parameters.WMI.Buffer**.

A driver can register or update a block with either static or dynamic instance names. Static instance names provide best performance; however, dynamic instance names are preferred for data blocks if the number of instances or instance names change frequently. For more information about instance names, see Defining WMI Instance Names.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IRP_MN_REGINFO

IRP_MN_REGINFO_EX

WMIREGINFO

WMIREGINFOW structure

1/25/2019 • 2 minutes to read • Edit Online

The **WMIREGINFO** structure contains information provided by a driver to register or update its data blocks and event blocks.

Syntax

```
typedef struct {
  ULONG       BufferSize;
  ULONG       NextWmiRegInfo;
  ULONG       RegistryPath;
  ULONG       MofResourceName;
  ULONG       GuidCount;
  WMIREGGUIDW WmiRegGuid[];
} WMIREGINFOW, *PWMIREGINFOW;
```

Members

`BufferSize`

Indicates the total size of the WMI registration data associated with this **WMIREGINFO** structure, calculated as follows: (**sizeof**(**WMIREGINFO**) + (**GuidCount** * **sizeof**(**WMIREGGUID**) + *additionaldata*). Additional data might include items such as the MOF resource name, registry path, and static instance names for blocks.

`NextWmiRegInfo`

If a driver handles WMI requests on behalf of another driver, as a class driver might on behalf of a miniclass driver, **NextWmiRegInfo** indicates the offset in bytes from the beginning of this **WMIREGINFO** to the next **WMIREGINFO** structure that contains WMI registration information for the other driver. Otherwise, **NextWmiRegInfo** is zero.

`RegistryPath`

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the registry path passed to the driver's DriverEntry routine. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIREGISTER).

`MofResourceName`

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the name of the MOF resource in the driver's image file. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIREGISTER).

`GuidCount`

Indicates the number of WMIREGGUID structures in the array at **WmiRegGuid**.

`WmiRegGuid`

Is an array of **GuidCount** WMIREGGUID structures.

Remarks

In response to a registration request (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIREGISTER), a driver builds at least one **WMIREGINFO** structure and writes the **WMIREGINFO** structure to the buffer at **IrpStack->Parameters.WMI.Buffer**. The **WMIREGINFO** structure contains an array of **WMIREGGUID** structures, one for each data block or event block exposed by the driver.

If the driver handles WMI requests on behalf of another driver, it builds another **WMIREGINFO** containing an array of **WMIREGGUID** structures for each block exposed by the other driver, sets the **NextWmiRegInfo** member of the first **WMIREGINFO** to an offset in bytes from the beginning of the first **WMIREGINFO** to the beginning of the next **WMIREGINFO** in the buffer, and writes both structures to the buffer. The driver indicates the total size of both **WMIREGINFO** structures and associated data when calls **IoCompleteRequest** to complete the IRP.

A driver can use the same **WMIREGINFO** structure(s) to remove or update blocks in response to an update request (IRP_MN_REGINFO or IRP_MN_REGINFO_EX with **Parameters.WMI.DataPath** set to WMIUPDATE). If WMIREG_FLAG_REMOVE_GUID is set in the **Flags** member of a **WMIREGGUID**, WMI removes that block from the list of blocks previously registered by the driver. If WMIREG_FLAG_REMOVE_GUID is clear, WMI updates registration information for that block only if other **WMIREGGUID** members have changed—otherwise, WMI does not change to its registration information for that block.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IRP_MN_REGINFO

IRP_MN_REGINFO_EX

IoCompleteRequest

WMIREGGUID

WNODE_HEADER structure

1/30/2019 • 6 minutes to read • Edit Online

The **WNODE_HEADER** structure is the first member of all other **WNODE_*XXX*** structures. It contains information common to all such structures.

Syntax

```
typedef struct _WNODE_HEADER {
  ULONG BufferSize;
  ULONG ProviderId;
  union {
    ULONG64 HistoricalContext;
    struct {
      ULONG Version;
      ULONG Linkage;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  union {
    ULONG         CountLost;
    HANDLE        KernelHandle;
    LARGE_INTEGER TimeStamp;
  } DUMMYUNIONNAME2;
  GUID  Guid;
  ULONG ClientContext;
  ULONG Flags;
} WNODE_HEADER, *PWNODE_HEADER;
```

Members

`BufferSize`

This member specifies the size, in bytes, of the nonpaged buffer to receive any **WNODE_*XXX*** data to be returned, including this **WNODE_HEADER** structure, additional members of a **WNODE_*XXX*** structure of the type indicated by **Flags**, and any WMI- or driver-determined data that accompanies that structure.

`ProviderId`

If **Flags** is set to WNODE_FLAG_EVENT_ITEM or WNODE_FLAG_EVENT_REFERENCE, **ProviderId** should contain the ID of the WMI provider associated with the device object. You can obtain the **ProviderId** value by calling IoWMIDeviceObjectToProviderId. If **Flags** is set to any other value, this member is reserved.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.HistoricalContext`

This member stores the handle to the event tracing session.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Version`

This member is reserved for WMI.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Linkage`

This member is reserved for WMI.

`DUMMYUNIONNAME2`

`DUMMYUNIONNAME2.CountLost`

Reserved

`DUMMYUNIONNAME2.KernelHandle`

This member is reserved for WMI.

`DUMMYUNIONNAME2.TimeStamp`

This member indicates the time at which a driver collected the **WNODE_*XXX*** data. This time value is expressed in absolute system time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601 in the Gregorian calendar. A driver can call **KeQuerySystemTime** to obtain this value. If the block is to be written to a log file (WNODE_FLAG_LOG_WNODE), an NT driver might also set WNODE_FLAG_USE_TIMESTAMP in **Flags** to request that the system logger leave the value of **TimeStamp** unchanged.

`Guid`

This member indicates the GUID that represents the data block associated with the **WNODE_*XXX*** to be returned.

`ClientContext`

This member stores the clock type for the session. Possible values are included in the following table.

VALUE	DESCRIPTION
1	Performance counter value (also called QPC, QueryPerformanceCounter, or PerfCounter)
2	System timer
3	CPU cycle

`Flags`

This member indicates the type of **WNODE_*XXX*** structure that contains the WNODE_HEADER structure:

WNODE_FLAG_ALL_DATA

The rest of a WNODE_ALL_DATA structure follows the **WNODE_HEADER** structure in the buffer.

WMI sets this flag in the **WNODE_HEADER** structure that it passes with an IRP_MN_QUERY_ALL_DATA request.

A driver sets this flag in the **WNODE_HEADER** structure of an event that consists of all instances of a data block. If the data block size is identical for all instances, a driver also sets WNODE_FLAG_FIXED_INSTANCE_SIZE.

WNODE_FLAG_EVENT_ITEM

A driver sets this flag to indicate that the **WNODE_*XXX*** structure was generated as an event. This flag is valid only if WNODE_FLAG_ALL_DATA, WNODE_FLAG_SINGLE_INSTANCE, or WNODE_FLAG_SINGLE_ITEM is also set.

WNODE_FLAG_EVENT_REFERENCE

The rest of a WNODE_EVENT_REFERENCE structure follows the **WNODE_HEADER** structure in the buffer.

A driver sets this flag when it generates an event that is larger than the maximum size specified in the registry for an event. WMI uses the information in the **WNODE_EVENT_REFERENCE** structure to request the event data and schedules such a request according to the value of WNODE_FLAG_SEVERITY_MASK.

WNODE_FLAG_METHOD_ITEM

The rest of a WNODE_METHOD_ITEM structure follows the **WNODE_HEADER** structure in the buffer.

WMI sets this flag in the **WNODE_HEADER** structure that it passes with an IRP_MN_EXECUTE_METHOD request.

WNODE_FLAG_SINGLE_INSTANCE

The rest of a WNODE_SINGLE_INSTANCE structure follows the **WNODE_HEADER** structure in the buffer.

WMI sets this flag in the **WNODE_HEADER** structure that it passes with a request to query or change an instance.

A driver sets this flag in the **WNODE_HEADER** structure of an event that consists of a single instance of a data block.

WNODE_FLAG_SINGLE_ITEM

The rest of a WNODE_SINGLE_INSTANCE structure follows the **WNODE_HEADER** structure in the buffer.

WMI sets this flag in the **WNODE_HEADER** structure that it passes with a request to change an item.

A driver sets this flag in the **WNODE_HEADER** structure of an event that consists of a single data item.

WNODE_FLAG_TOO_SMALL

The rest of a WNODE_TOO_SMALL structure follows the **WNODE_HEADER** structure in the buffer.

A driver sets this flag when it passes a **WNODE_TOO_SMALL** structure, indicating that the buffer is too small for all of the **WNODE_*XXX*** data to be returned.

In addition, **Flags** might be set with one or more of the following flags that provide additional information about the **WNODE_*XXX***:

WNODE_FLAG_FIXED_INSTANCE_SIZE

All instances of a data block are the same size. This flag is valid only if WNODE_FLAG_ALL_DATA is also set.

WNODE_FLAG_INSTANCES_SAME

The number of instances and the dynamic instance names in a **WNODE_ALL_DATA** structure to be returned are identical to those returned from the previous **WNODE_ALL_DATA** query. This flag is valid only if WNODE_FLAG_ALL_DATA is also set. This flag is ignored for data blocks registered with static instance names.

For optimized performance, a driver should set this flag if it can track changes to the number or names of its data blocks. WMI can then skip the processing required to detect and update dynamic instance names.

WNODE_FLAG_STATIC_INSTANCE_NAMES

The **WNODE_*XXX*** data to be returned does not include instance names.

WMI sets this flag before requesting **WNODE_*XXX*** data for data blocks registered with static instance names. After receiving the returned **WNODE_*XXX*** from the driver, WMI fills in the static instance names specified at registration before passing the returned **WNODE_*XXX*** to a data consumer.

WNODE_FLAG_PDO_INSTANCE_NAMES

Static instance names are based on the device instance ID of the PDO for the device. A driver requests such names by setting WMIREG_FLAG_INSTANCE_PDO in the WMIREGGUID it uses to register the block.

WMI sets this flag before requesting **WNODE_*XXX*** data for data blocks registered with PDO-based instance names.

WNODE_FLAG_SEVERITY_MASK

The driver-determined severity level of the event associated with a returned WNODE_EVENT_REFERENCE, with 0x00 indicating the least severe and 0xff indicating the most severe level.

WMI uses the value of this flag to prioritize its requests for the event data.

WNODE_FLAG_USE_TIMESTAMP

The system logger should not modify the value of **TimeStamp** set by the driver.

An NT driver might also set **Flags** to one or more of the following values for event blocks to be written to a system log file:

WNODE_FLAG_LOG_WNODE

An event block is to be sent to the system logger. The event header is a standard **WNODE_HEADER** structure. If the driver clears WNODE_FLAG_TRACED_GUID, the block will also be sent to WMI for delivery to any data consumers that have enabled the event. The driver must allocate the **WNODE_*XXX*** from pool memory. WMI frees the memory after delivering the event to data consumers.

WNODE_FLAG_TRACED_GUID

An event block is to be sent only to the system logger. It does not get sent to WMI data consumers. The event header is an **EVENT_TRACE_HEADER** structure, declared in *Evntrace.h*, instead of a **WNODE_HEADER**. The driver must allocate memory for the **WNODE_*XXX*** and free it after IoWMIWriteEvent returns. The driver can allocate such memory either from the stack or, to minimize the overhead of allocating and freeing the memory, from the driver's thread local storage if the driver creates and maintains its own thread pool.

WNODE_FLAG_USE_GUID_PTR

The **Guid** member points to a GUID in memory, rather than containing the GUID itself. The system logger dereferences the pointer before passing the data to the consumer. This flag is valid only if WNODE_FLAG_LOG_WNODE or WNODE_FLAG_TRACED_GUID are also set.

WNODE_FLAG_USE_MOF_PTR

Data that follows the fixed members of a **WNODE_*XXX*** structure consists of an array of MOF_FIELD structures, defined in Evntrace.h, that contain pointers to data and sizes rather than the data itself. The array can contain up to MAX_MOF_FIELD elements. The system logger dereferences the pointers before passing the data to the consumer This flag is valid only for blocks registered with WMIREG_FLAG_TRACED_GUID.

Remarks

In an **IRP_MN_CHANGE_*XXX*** or IRP_MN_EXECUTE_METHOD request, **BufferSize** in the IRP indicates the maximum size in bytes of the output buffer, while **BufferSize** in the input **WNODE_HEADER** for such a request indicates the size, in bytes, of the input data in the buffer.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IoWMIDeviceObjectToProviderId

IoWMIWriteEvent

KeQuerySystemTime

WNODE_ALL_DATA

WNODE_EVENT_ITEM

WNODE_EVENT_REFERENCE

WNODE_METHOD_ITEM

WNODE_SINGLE_INSTANCE

WNODE_SINGLE_ITEM

WNODE_TOO_SMALL

WNODE_ALL_DATA structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_ALL_DATA** structure contains data for all instances of a data block or event block.

Syntax

```
typedef struct tagWNODE_ALL_DATA {
  struct _WNODE_HEADER WnodeHeader;
  ULONG                DataBlockOffset;
  ULONG                InstanceCount;
  ULONG                OffsetInstanceNameOffsets;
  union {
    ULONG                       FixedInstanceSize;
    OFFSETINSTANCEDATAANDLENGTH OffsetInstanceDataAndLength[];
  } DUMMYUNIONNAME;
} WNODE_ALL_DATA, *PWNODE_ALL_DATA;
```

Members

`WnodeHeader`

Specifies a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`DataBlockOffset`

Indicates the offset in bytes from the beginning of the **WNODE_ALL_DATA** structure to the beginning of data for the first instance.

`InstanceCount`

Indicates the number of instances whose data follows the fixed members of the **WNODE_ALL_DATA** in the buffer at **IrpStack->Parameters.WMI.Buffer**.

`OffsetInstanceNameOffsets`

Indicates the offset in bytes from the beginning of the **WNODE_ALL_DATA** to an array of offsets to dynamic instance names. Each instance name must be aligned on a USHORT boundary. If all instances to be returned have static instance names, WMI ignores **OffsetInstanceNameOffsets**.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.FixedInstanceSize`

Indicates the size of each instance to be returned if all such instances are the same size. This member is valid only if the driver sets WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags**.

`DUMMYUNIONNAME.OffsetInstanceDataAndLength`

If instances to be returned vary in size, **OffsetInstanceDataAndLength** is an array of **InstanceCount** **OFFSETINSTANCEDATAANDLENGTH** structures that specify the offset in bytes from the beginning of the **WNODE_ALL_DATA** to the beginning of each instance and its length. **OFFSETINSTANCEDATAANDLENGTH** is defined as follows:

```
typedef struct {
  ULONG  OffsetInstanceData;
  ULONG  LengthInstanceData;
} OFFSETINSTANCEDATAANDLENGTH, *POFFSETINSTANCEDATAANDLENGTH;
```

Each instance must be aligned on a USHORT boundary. The **OffsetInstanceDataAndLength** member is valid only if the driver clears WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags**.

Remarks

A driver fills in a **WNODE_ALL_DATA** structure in response to an IRP_MN_QUERY_ALL_DATA request. A driver might also generate a **WNODE_ALL_DATA** as an event.

After filling in the fixed members of the structure, a driver writes instance data and dynamic instance names (if any) at **DataBlockOffset** and **OffsetInstanceNameOffsets**, respectively, in the buffer at **IrpStack->Parameters.WMI.Buffer**. If WNODE_FLAG_FIXED_INSTANCE_SIZE is clear, the first offset follows the last element of the **OffsetInstanceDataAndLength** array, plus padding so the data begins on an 8-byte boundary.

Instance names must be USHORT aligned. Instance data must be QUADWORD aligned.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IRP_MN_QUERY_ALL_DATA

WNODE_EVENT_ITEM

WNODE_HEADER

WNODE_EVENT_ITEM structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_EVENT_ITEM** structure contains data generated by a driver for an event.

Syntax

```
typedef struct tagWNODE_EVENT_ITEM {
  struct _WNODE_HEADER WnodeHeader;
} WNODE_EVENT_ITEM, *PWNODE_EVENT_ITEM;
```

Members

`WnodeHeader`

Specifies a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

Remarks

The **WnodeHeader** member of the **WNODE_EVENT_ITEM** structure is followed by a structure whose type depends on the flags that are set in **WnodeHeader**. Possibilities include WNODE_ALL_DATA, WNODE_SINGLE_INSTANCE, and WNODE_SINGLE_ITEM. For more information about the flags, see WNODE_HEADER.

The **ProviderId** member of the WNODE_HEADER structure for use in a **WNODE_EVENT_ITEM** structure should be initialized using IoWMIDeviceObjectToProviderId.

A driver generates only events that it has previously enabled in response to an IRP_MN_ENABLE_EVENTS request. To generate an event, a driver calls IoWMIWriteEvent and passes a pointer to the **WNODE_EVENT_ITEM**. WMI queues the event for delivery to all data consumers registered for that event.

For best performance, events should be small in size. However, if the amount of data for an event exceeds the maximum size defined in the registry, a driver can pass a WNODE_EVENT_REFERENCE, which WMI uses to query for the related **WNODE_EVENT_ITEM**. For more information about defining and generating WMI events, see Windows Management Instrumentation.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IRP_MN_ENABLE_EVENTS

IoWMIDeviceObjectToProviderId

IoWMIWriteEvent

WNODE_ALL_DATA

WNODE_EVENT_REFERENCE

WNODE_HEADER

WNODE_SINGLE_INSTANCE

WNODE_SINGLE_ITEM

WNODE_EVENT_REFERENCE structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_EVENT_REFERENCE** structure contains information that WMI can use to query for an event that exceeds the event size limit set in the registry.

Syntax

```
typedef struct tagWNODE_EVENT_REFERENCE {
  struct _WNODE_HEADER WnodeHeader;
  GUID                 TargetGuid;
  ULONG                TargetDataBlockSize;
  union {
    ULONG TargetInstanceIndex;
    WCHAR TargetInstanceName[];
  } DUMMYUNIONNAME;
} WNODE_EVENT_REFERENCE, *PWNODE_EVENT_REFERENCE;
```

Members

`WnodeHeader`

Is a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the provider ID, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`TargetGuid`

Indicates the GUID that represents the event to query.

`TargetDataBlockSize`

Indicates the size of the event.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.TargetInstanceIndex`

Indicates the index into the driver's list of static instance names for the event. This member is valid only if the event block was registered with static instance names and WNODE_FLAGS_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**.

`DUMMYUNIONNAME.TargetInstanceName`

Indicates the dynamic instance name of the event as a counted Unicode string. This member is valid only if WNODE_FLAGS_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags** and the event block was registered with dynamic instance names.

Remarks

If the amount of data for an event exceeds the maximum size set in the registry, a driver can generate a **WNODE_EVENT_REFERENCE** that specifies a WNODE_EVENT_ITEM that WMI can query to obtain the event. For more information about defining and generating WMI events, see Implementing WMI.

The **ProviderId** member of the WNODE_HEADER structure for use in a **WNODE_EVENT_REFERENCE** structure should be initialized using IoWMIDeviceObjectToProviderId.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

IoWMIDeviceObjectToProviderId

WNODE_EVENT_ITEM

WNODE_HEADER

WNODE_METHOD_ITEM structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_METHOD_ITEM** structure indicates a method associated with an instance of a data block and contains any input data for the method.

Syntax

```
typedef struct tagWNODE_METHOD_ITEM {
  struct _WNODE_HEADER WnodeHeader;
  ULONG                OffsetInstanceName;
  ULONG                InstanceIndex;
  ULONG                MethodId;
  ULONG                DataBlockOffset;
  ULONG                SizeDataBlock;
  UCHAR                VariableData[];
} WNODE_METHOD_ITEM, *PWNODE_METHOD_ITEM;
```

Members

`WnodeHeader`

Is a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`OffsetInstanceName`

Indicates the offset in bytes from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

`InstanceIndex`

Indicates the index of this instance into the driver's list of static instance names for this data block. This member is valid only if the data block was registered with static instance names and WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

`MethodId`

Specifies the ID of the method to run.

`DataBlockOffset`

Indicates the offset from the beginning of an input **WNODE_METHOD_ITEM** to input data for the method, or the offset from the beginning of an output **WNODE_METHOD_ITEM** to output data from the method.

`SizeDataBlock`

Indicates the size of the input data in an input **WNODE_METHOD_ITEM**, or zero if there is no input. In an output **WNODE_METHOD_ITEM**, **SizeDataBlock** indicates the size of the output data, or zero if there is no output.

`VariableData`

Contains additional data, including the dynamic instance name if any, and the input for or output from the method aligned on an 8-byte boundary.

Remarks

WMI passes a **WNODE_METHOD_ITEM** with an IRP_MN_EXECUTE_METHOD request to specify a method to run in an instance of a data block, plus any input data required by the method.

If a method generates output, a driver overwrites the input data with the output at **DataBlockOffset** in the buffer at **IrpStack->Parameters.WMI.Buffer**, and sets **SizeDataBlock** in the **WNODE_METHOD_ITEM** to specify the size of the output data.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

WNODE_HEADER

WNODE_SINGLE_INSTANCE structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_SINGLE_INSTANCE** structure contains values for all data items in one instance of a data block.

Syntax

```
typedef struct tagWNODE_SINGLE_INSTANCE {
  struct _WNODE_HEADER WnodeHeader;
  ULONG                OffsetInstanceName;
  ULONG                InstanceIndex;
  ULONG                DataBlockOffset;
  ULONG                SizeDataBlock;
  UCHAR                VariableData[];
} WNODE_SINGLE_INSTANCE, *PWNODE_SINGLE_INSTANCE;
```

Members

`WnodeHeader`

Is a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`OffsetInstanceName`

Indicates the offset from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

`InstanceIndex`

Indicates the index of an instance registered with static instance names. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

`DataBlockOffset`

Indicates the offset from the beginning of this structure to the beginning of the instance.

`SizeDataBlock`

Indicates the size of the data block for this instance.

`VariableData`

Contains additional data, including the dynamic instance name if any, padding so the instance begins on an 8-byte boundary, and the instance of the data block to be returned.

Remarks

WMI passes a **WNODE_SINGLE_INSTANCE** with an IRP_MN_CHANGE_SINGLE_INSTANCE request to set read/write data items in an instance of a data block. A driver can ignore values passed for read-only data items in

the instance.

A driver fills in a **WNODE_SINGLE_INSTANCE** in response to an IRP_MN_QUERY_SINGLE_INSTANCE request or to generate an event that consists of a single instance.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

WNODE_EVENT_ITEM

WNODE_HEADER

WNODE_SINGLE_ITEM structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_SINGLE_ITEM** structure contains the value of a single data item in an instance of a data block.

Syntax

```
typedef struct tagWNODE_SINGLE_ITEM {
  struct _WNODE_HEADER WnodeHeader;
  ULONG                OffsetInstanceName;
  ULONG                InstanceIndex;
  ULONG                ItemId;
  ULONG                DataBlockOffset;
  ULONG                SizeDataItem;
  UCHAR                VariableData[];
} WNODE_SINGLE_ITEM, *PWNODE_SINGLE_ITEM;
```

Members

`WnodeHeader`

Is a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`OffsetInstanceName`

Indicates the offset from the beginning of this structure to the dynamic instance name, if any, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

`InstanceIndex`

Indicates the index into the driver's list of static instance names of this instance. This member is valid only if the data block was registered with static instance names and WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

`ItemId`

Specifies the ID of the data item to set.

`DataBlockOffset`

Indicates the offset from the beginning of this structure to the new value for the data item.

`SizeDataItem`

Indicates the size of the data item.

`VariableData`

Contains additional data, including the dynamic instance name if any, padding so the data value begins on an 8-byte boundary, and the new value for the data item.

Remarks

WMI passes a **WNODE_SINGLE_ITEM** with an IRP_MN_CHANGE_SINGLE_ITEM request to set the value of a data item in an instance of a data block.

A driver builds a **WNODE_SINGLE_ITEM** to generate an event that consists of a single data item.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

WNODE_EVENT_ITEM

WNODE_HEADER

WNODE_TOO_SMALL structure

1/25/2019 • 2 minutes to read • Edit Online

The **WNODE_TOO_SMALL** structure indicates the size of the buffer needed to receive output from a request.

Syntax

```
typedef struct tagWNODE_TOO_SMALL {
  struct _WNODE_HEADER WnodeHeader;
  ULONG                SizeNeeded;
} WNODE_TOO_SMALL, *PWNODE_TOO_SMALL;
```

Members

`WnodeHeader`

Is a WNODE_HEADER structure that contains information common to all **WNODE_*XXX*** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_*XXX*** data being passed or returned.

`SizeNeeded`

Specifies the size of the buffer needed to receive all of the **WNODE_*XXX*** data to be returned.

Remarks

When the buffer for a WMI request is too small to receive all of the data to be returned, a driver fills in a **WNODE_TOO_SMALL** structure to indicate the required buffer size. WMI can then increase the buffer to the recommended size and issue the request again. A driver is responsible for managing any side effects caused by handling the same request more than once.

Requirements

Header	wmistr.h (include Wmistr.h)

See Also

WNODE_HEADER

wudfwdm.h header

1/8/2019 • 6 minutes to read • Edit Online

This header is used by wdf. For more information, see:

- Windows Driver Framework wudfwdm.h contains the following programming interfaces:

Functions

TITLE	DESCRIPTION
AppendTailList	The AppendTailList routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of LIST_ENTRY structures.
ARGUMENT_PRESENT	The ARGUMENT_PRESENT macro takes an argument pointer and returns FALSE if the pointer is NULL. Otherwise, it returns TRUE.
InitializeListHead	The InitializeListHead routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.
InitializeObjectAttributes	The InitializeObjectAttributes macro initializes the opaque OBJECT_ATTRIBUTES structure, which specifies the properties of an object handle to routines that open handles.
InsertHeadList	The InsertHeadList routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.
InsertTailList	The InsertTailList routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.
IsListEmpty	The IsListEmpty routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.
PopEntryList	The PopEntryList routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.
PushEntryList	The PushEntryList routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.
READ_REGISTER_BUFFER_UCHAR	The READ_REGISTER_BUFFER_UCHAR routine reads a number of bytes from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG	The READ_REGISTER_BUFFER_ULONG routine reads a number of ULONG values from the specified register address into a buffer.
READ_REGISTER_BUFFER_ULONG64	The READ_REGISTER_BUFFER_ULONG64 function reads a number of ULONG64 values from the specified register address into a buffer.

TITLE	DESCRIPTION
READ_REGISTER_BUFFER_USHORT	The READ_REGISTER_BUFFER_USHORT routine reads a number of USHORT values from the specified register address into a buffer.
READ_REGISTER_UCHAR	The READ_REGISTER_UCHAR routine reads a byte from the specified register address.
READ_REGISTER_ULONG	The READ_REGISTER_ULONG routine reads a ULONG value from the specified register address.
READ_REGISTER_ULONG64	The READ_REGISTER_ULONG64 function reads a ULONG64 value from the specified register address.
READ_REGISTER_USHORT	The READ_REGISTER_USHORT routine reads a USHORT value from the specified register address.
RemoveEntryList	The RemoveEntryList routine removes an entry from a doubly linked list of LIST_ENTRY structures.
RemoveHeadList	The RemoveHeadList routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.
RemoveTailList	The RemoveTailList routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.
RtlAppendUnicodeStringToString	The RtlAppendUnicodeStringToString routine concatenates two Unicode strings.
RtlAppendUnicodeToString	The RtlAppendUnicodeToString routine concatenates the supplied Unicode string to a buffered Unicode string.
RtlCompareUnicodeString	The RtlCompareUnicodeString routine compares two Unicode strings.
RtlCopyUnicodeString	The RtlCopyUnicodeString routine copies a source string to a destination string.
RtlEqualUnicodeString	The RtlEqualUnicodeString routine compares two Unicode strings to determine whether they are equal.
RtlFreeUnicodeString	The RtlFreeUnicodeString routine releases storage that was allocated by RtlAnsiStringToUnicodeString or RtlUpcaseUnicodeString.
RtlInitUnicodeString	For more information, see the WdmlibRtlInitUnicodeStringEx function.
RtlInt64ToUnicodeString	The RtlInt64ToUnicodeString routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.
RtlIntegerToUnicodeString	The RtlIntegerToUnicodeString routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.

TITLE	DESCRIPTION
RtlIntPtrToUnicodeString	The RtlIntPtrToUnicodeString routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.
RtlUnicodeStringToInteger	The RtlUnicodeStringToInteger routine converts a Unicode string representation of a number to the equivalent integer value.
WRITE_REGISTER_BUFFER_UCHAR	The WRITE_REGISTER_BUFFER_UCHAR routine writes a number of bytes from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG	The WRITE_REGISTER_BUFFER_ULONG routine writes a number of ULONG values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_ULONG64	The WRITE_REGISTER_BUFFER_ULONG64 function writes a number of ULONG64 values from a buffer to the specified register.
WRITE_REGISTER_BUFFER_USHORT	The WRITE_REGISTER_BUFFER_USHORT routine writes a number of USHORT values from a buffer to the specified register.
WRITE_REGISTER_UCHAR	The WRITE_REGISTER_UCHAR routine writes a byte to the specified address.
WRITE_REGISTER_ULONG	The WRITE_REGISTER_ULONG routine writes a ULONG value to the specified address.
WRITE_REGISTER_ULONG64	The WRITE_REGISTER_ULONG64 function writes a ULONG64 value to the specified address.
WRITE_REGISTER_USHORT	The WRITE_REGISTER_USHORT routine writes a USHORT value to the specified address.

Callback functions

TITLE	DESCRIPTION
DRIVER_INITIALIZE	DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.
PINTERFACE_DEREFERENCE	The InterfaceDereference routine decrements the reference count on a driver-defined interface.
PINTERFACE_REFERENCE	The InterfaceReference routine increments the reference count on a driver-defined interface.
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK	The ComponentActiveConditionCallback callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.

TITLE	DESCRIPTION
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK	The ComponentCriticalTransitionCallback callback routine handles a transition of the specified component between the F0 (fully on) and low-power Fx component power states.
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK	The ComponentIdleConditionCallback callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
PO_FX_COMPONENT_IDLE_STATE_CALLBACK	The ComponentIdleStateCallback callback routine notifies the driver of a pending change to the Fx power state of the specified component.
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK	The DevicePowerNotRequiredCallback callback routine notifies the device driver that the device is not required to stay in the D0 power state.
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK	The DevicePowerRequiredCallback callback routine notifies the device driver that the device must enter and remain in the D0 power state.
PO_FX_POWER_CONTROL_CALLBACK	The PowerControlCallback callback routine performs a power control operation that is requested by the power management framework (PoFx).

Structures

TITLE	DESCRIPTION
IO_STATUS_BLOCK	A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.
CM_FULL_RESOURCE_DESCRIPTOR	The CM_FULL_RESOURCE_DESCRIPTOR structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.
CM_PARTIAL_RESOURCE_DESCRIPTOR	The CM_PARTIAL_RESOURCE_DESCRIPTOR structure specifies one or more system hardware resources, of a single type, assigned to a device.
CM_PARTIAL_RESOURCE_LIST	The CM_PARTIAL_RESOURCE_LIST structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.
CM_RESOURCE_LIST	The CM_RESOURCE_LIST structure specifies all of the system hardware resources assigned to a device.
OBJECT_ATTRIBUTES	The OBJECT_ATTRIBUTES structure specifies attributes that can be applied to objects or object handles by routines that create objects and/or return handles to objects.

TITLE	DESCRIPTION
PO_FX_COMPONENT_IDLE_STATE	The PO_FX_COMPONENT_IDLE_STATE structure specifies the attributes of an Fx power state of a component in a device.
PO_FX_COMPONENT_PERF_INFO	The PO_FX_COMPONENT_PERF_INFO structure describes all the sets of performance states for a single component within a device.
PO_FX_COMPONENT_PERF_SET	The PO_FX_COMPONENT_PERF_SET structure represents a set of performance states for a single component within a device.
PO_FX_COMPONENT_V1	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_V2	The PO_FX_COMPONENT structure describes the power state attributes of a component in a device.
PO_FX_DEVICE_V1	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_DEVICE_V2	The PO_FX_DEVICE structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_PERF_STATE	The PO_FX_PERF_STATE structure represents a performance state for a single component within a device.
PO_FX_PERF_STATE_CHANGE	The PO_FX_PERF_STATE_CHANGE structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.
UNICODE_STRING	The UNICODE_STRING structure is used to define Unicode strings.

Enumerations

TITLE	DESCRIPTION
DEVICE_REGISTRY_PROPERTY	The DEVICE_REGISTRY_PROPERTY enumeration identifies device properties that are stored in the registry.
INTERFACE_TYPE	The INTERFACE_TYPE enumeration indicates the bus type.
KINTERRUPT_MODE	The KINTERRUPT_MODE enumeration type indicates whether an interrupt is level-triggered or edge-triggered.
MEMORY_CACHING_TYPE	The MEMORY_CACHING_TYPE enumeration type specifies the permitted caching behavior when allocating or mapping memory.
PO_FX_PERF_STATE_TYPE	The PO_FX_PERF_STATE_TYPE enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.

TITLE	DESCRIPTION
PO_FX_PERF_STATE_UNIT	The PO_FX_PERF_STATE_UNIT enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.
POOL_TYPE	The POOL_TYPE enumeration type specifies the type of system memory to allocate.

CM_FULL_RESOURCE_DESCRIPTOR structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_FULL_RESOURCE_DESCRIPTOR** structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.

Syntax

```
typedef struct _CM_FULL_RESOURCE_DESCRIPTOR {
  INTERFACE_TYPE           InterfaceType;
  ULONG                    BusNumber;
  CM_PARTIAL_RESOURCE_LIST PartialResourceList;
} CM_FULL_RESOURCE_DESCRIPTOR, *PCM_FULL_RESOURCE_DESCRIPTOR;
```

Members

`InterfaceType`

Specifies the type of bus to which the device is connected. This must be one of the types defined by INTERFACE_TYPE, in Wdm.h or Ntddk.h. (Not used by WDM drivers.)

`BusNumber`

The system-assigned, driver-supplied, zero-based number of the bus to which the device is connected. (Not used by WDM drivers.)

`PartialResourceList`

A CM_PARTIAL_RESOURCE_LIST structure.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_PARTIAL_RESOURCE_DESCRIPTOR structure

1/30/2019 • 9 minutes to read • Edit Online

The **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure specifies one or more system hardware resources, of a single type, assigned to a device. This structure is used to create an array within a CM_PARTIAL_RESOURCE_LIST structure.

Syntax

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
  UCHAR  Type;
  UCHAR  ShareDisposition;
  USHORT Flags;
  union {
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Generic;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Port;
    struct {
#if ...
      USHORT    Level;
      USHORT    Group;
#else
      ULONG     Level;
#endif
      ULONG     Vector;
      KAFFINITY Affinity;
    } Interrupt;
    struct {
      union {
        struct {
          USHORT    Group;
          USHORT    Reserved;
          USHORT    MessageCount;
          ULONG     Vector;
          KAFFINITY Affinity;
        } Raw;
        struct {
#if ...
          USHORT    Level;
          USHORT    Group;
#else
          ULONG     Level;
#endif
          ULONG     Vector;
          KAFFINITY Affinity;
        } Translated;
      } DUMMYUNIONNAME;
    } MessageInterrupt;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length;
    } Memory;
    struct {
      ULONG Channel;
      ULONG Port;
      ULONG Reserved1;
```

```
      ULONG Reserved1;
    } Dma;
    struct {
      ULONG Channel;
      ULONG RequestLine;
      UCHAR TransferWidth;
      UCHAR Reserved1;
      UCHAR Reserved2;
      UCHAR Reserved3;
    } DmaV3;
    struct {
      ULONG Data[3];
    } DevicePrivate;
    struct {
      ULONG Start;
      ULONG Length;
      ULONG Reserved;
    } BusNumber;
    struct {
      ULONG DataSize;
      ULONG Reserved1;
      ULONG Reserved2;
    } DeviceSpecificData;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length40;
    } Memory40;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length48;
    } Memory48;
    struct {
      PHYSICAL_ADDRESS Start;
      ULONG            Length64;
    } Memory64;
    struct {
      UCHAR Class;
      UCHAR Type;
      UCHAR Reserved1;
      UCHAR Reserved2;
      ULONG IdLowPart;
      ULONG IdHighPart;
    } Connection;
  } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR, *PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

Members

`Type`

Identifies the resource type. The constant value specified for **Type** indicates which structure within the **u** union is valid, as indicated in the following table. (These flags are used within both **CM_PARTIAL_RESOURCE_DESCRIPTOR** and IO_RESOURCE_DESCRIPTOR structures, except where noted.)

TYPE VALUE	U MEMBER SUBSTRUCTURE
CmResourceTypePort	**u.Port**
CmResourceTypeInterrupt	**u.Interrupt** or **u.MessageInterrupt**.If the CM_RESOURCE_INTERRUPT_MESSAGE flag of **Flags** is set, use **u.MessageInterrupt**; otherwise, use **u.Interrupt**.
CmResourceTypeMemory	**u.Memory**

CmResourceTypeMemoryLarge	One of **u.Memory40**, **u.Memory48**, or **u.Memory64**.The CM_RESOURCE_MEMORY_LARGE_*XXX* flags set in the **Flags** member determines which structure is used.
CmResourceTypeDma	**u.Dma** (if CM_RESOURCE_DMA_V3 is not set) or **u.DmaV3** (if CM_RESOURCE_DMA_V3 flag is set)
CmResourceTypeDevicePrivate	**u.DevicePrivate**
CmResourceTypeBusNumber	**u.BusNumber**
CmResourceTypeDeviceSpecific	**u.DeviceSpecificData**(Not used within **IO_RESOURCE_DESCRIPTOR**.)
CmResourceTypePcCardConfig	**u.DevicePrivate**
CmResourceTypeMfCardConfig	**u.DevicePrivate**
CmResourceTypeConnection	**u.Connection**
CmResourceTypeConfigData	*Reserved for system use.*
CmResourceTypeNonArbitrated	*Not used.*

`ShareDisposition`

Indicates whether the described resource can be shared. Valid constant values are listed in the following table.

VALUE	MEANING
CmResourceShareDeviceExclusive	The device requires exclusive use of the resource.
CmResourceShareDriverExclusive	The driver requires exclusive use of the resource. (Not supported for WDM drivers.)

CmResourceShareShared	The resource can be shared without restriction.

`Flags`

Contains flag bits that are specific to the resource type, as indicated in the following table. Flags can be bitwise-ORed together as appropriate.

RESOURCE TYPE	FLAG	DEFINITION
CmResourceTypePort		
	CM_RESOURCE_PORT_MEMORY	The device is accessed in memory address space.
	CM_RESOURCE_PORT_IO	The device is accessed in I/O address space.
	CM_RESOURCE_PORT_10_BIT_DECODE	The device decodes 10 bits of the port address.
	CM_RESOURCE_PORT_12_BIT_DECODE	The device decodes 12 bits of the port address.
	CM_RESOURCE_PORT_16_BIT_DECODE	The device decodes 16 bits of the port address.
	CM_RESOURCE_PORT_POSITIVE_DECODE	The device uses "positive decode" instead of "subtractive decode". (In general, PCI devices use positive decode and ISA buses use subtractive decode.)
	CM_RESOURCE_PORT_PASSIVE_DECODE	The device decodes the port but the driver does not use it.
	CM_RESOURCE_PORT_WINDOW_DECODE	Reserved for system use.
CmResourceTypeInterrupt		
	CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE	The IRQ line is level-triggered. (These IRQs are usually sharable.)
	CM_RESOURCE_INTERRUPT_LATCHED	The IRQ line is edge-triggered.

	CM_RESOURCE_INTERRUPT_MESSAGE	If this flag is set, the interrupt is a message-signaled interrupt. Otherwise, the interrupt is a line-based interrupt. This flag can be set starting with Windows Vista.
	CM_RESOURCE_INTERRUPT_POLICY_INCLUDED	Not used with the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure. For more information about this flag, see IO_RESOURCE_DESCRIPTOR.
	CM_RESOURCE_INTERRUPT_SECONDARY_INTERRUPT	The interrupt is a secondary interrupt. This flag can be set starting with Windows 8. For more information about secondary interrupts, see GPIO Interrupts.
	CM_RESOURCE_INTERRUPT_WAKE_HINT	The interrupt is capable of waking the operating system from a low-power idle state or a system sleep state. This flag can be set starting with Windows 8. For more information about wake capabilities, see Enabling Device Wake-Up.
CmResourceTypeMemory		
	CM_RESOURCE_MEMORY_READ_WRITE	The memory range is readable and writable.
	CM_RESOURCE_MEMORY_READ_ONLY	The memory range is read-only.
	CM_RESOURCE_MEMORY_WRITE_ONLY	The memory range is write-only.
	CM_RESOURCE_MEMORY_PREFETCHABLE	The memory range is prefetchable.
	CM_RESOURCE_MEMORY_COMBINEDWRITE	Combined-write caching is allowed.
	CM_RESOURCE_MEMORY_24	The device uses 24-bit addressing.
	CM_RESOURCE_MEMORY_CACHEABLE	The memory range is cacheable.
CmResourceTypeMemoryLarge		

	CM_RESOURCE_MEMORY_LARGE_40	The memory descriptor uses the **u.Memory40** member.
	CM_RESOURCE_MEMORY_LARGE_48	The memory descriptor uses the **u.Memory48** member.
	CM_RESOURCE_MEMORY_LARGE_64	The memory descriptor uses the **u.Memory64** member.
CmResourceTypeDma		
	CM_RESOURCE_DMA_8	8-bit DMA channel
	CM_RESOURCE_DMA_16	16-bit DMA channel
	CM_RESOURCE_DMA_32	32-bit DMA channel
	CM_RESOURCE_DMA_8_AND_16	8-bit and 16-bit DMA channel
	CM_RESOURCE_DMA_BUS_MASTER	The device supports bus master DMA transfers.
	CM_RESOURCE_DMA_TYPE_A	Type A DMA
	CM_RESOURCE_DMA_TYPE_B	Type B DMA
	CM_RESOURCE_DMA_TYPE_F	Type F DMA
	CM_RESOURCE_DMA_V3	Use the **DmaV3** member instead of the **Dma** member. The **DmaV3** member is available starting with Windows 8.

`u`

`u.Generic`

`u.Generic.Start`

`u.Generic.Length`

`u.Port`

`u.Port.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Port.Length`

The length, in bytes, of the range of allocated I/O port addresses.

`u.Interrupt`

`u.Interrupt.Level.#1877845754`

`u.Interrupt.Group`

Specifies the processor group number. This member exists only if the NT_PROCESSOR_GROUPS constant is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Group** and **Affinity** members together specify a group affinity that indicates which processors the device can interrupt. To specify an affinity for any group, set **Group** to ALL_PROCESSOR_GROUPS.

`u.Interrupt.Level.#3336307498`

`u.Interrupt.Vector`

For raw resources: Specifies the device's bus-specific interrupt vector (if appropriate for the platform and bus).

For translated resources: Specifies the global system interrupt vector assigned to the device.

For more information about raw and translated resources, see Remarks.

`u.Interrupt.Affinity`

Contains a **KAFFINITY**-typed bitmask value indicating the set of processors the device can interrupt. To indicate that the device can interrupt any processor, this member is set to -1.

`u.MessageInterrupt`

`u.MessageInterrupt.DUMMYUNIONNAME`

`u.MessageInterrupt.DUMMYUNIONNAME.Raw`

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Group`

Specifies a processor group number. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Group** and **Affinity** members together specify a group affinity that indicates which processors can receive the device's interrupts. To specify an affinity for any group, set **Group** to ALL_PROCESSOR_GROUPS.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Reserved`

Not used.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.MessageCount`

Specifies the number of message-signaled interrupts generated for this driver.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Vector`

Specifies the device's interrupt vector.

`u.MessageInterrupt.DUMMYUNIONNAME.Raw.Affinity`

Specifies a KAFFINITY value that indicates the processors that receive the device's interrupts.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Level.#1877845754`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Group`

Specifies a processor group number. This member exists only if NT_PROCESSOR_GROUPS is defined at compile time. This member can be nonzero only on Windows 7 and later versions of Windows. The **Translated.Group** and

Translated.Affinity members together specify a group affinity that indicates which processors can receive the device's interrupts. To specify an affinity for any group, set **Translated.Group** to ALL_PROCESSOR_GROUPS.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Level.#3336307498`

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Vector`

Specifies the device's interrupt vector.

`u.MessageInterrupt.DUMMYUNIONNAME.Translated.Affinity`

Specifies a KAFFINITY value that identifies the processors that receive the device's interrupts.

`u.Memory`

`u.Memory.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory.Length`

Specifies the length, in bytes, of the range of allocated memory addresses.

`u.Dma`

`u.Dma.Channel`

Specifies the number of the DMA channel on a system DMA controller that the device can use.

`u.Dma.Port`

Specifies the number of the DMA port that an MCA-type device can use.

`u.Dma.Reserved1`

Not used.

`u.DmaV3`

`u.DmaV3.Channel`

Specifies the number of the DMA channel on the system DMA controller that is allocated to the device.

`u.DmaV3.RequestLine`

Specifies the number of the request line on the system DMA controller that is allocated to the device.

`u.DmaV3.TransferWidth`

Specifies the width, in bits, of the data bus that the system DMA controller that is allocated to the device uses to transfer data to or from the device.

`u.DmaV3.Reserved1`

Not used.

`u.DmaV3.Reserved2`

Not used.

`u.DmaV3.Reserved3`

Not used.

`u.DevicePrivate`

`u.DevicePrivate.Data`

`u.BusNumber`

`u.BusNumber.Start`

Specifies the lowest-numbered of a range of contiguous buses allocated to the device.

`u.BusNumber.Length`

Specifies the number of buses allocated to the device.

`u.BusNumber.Reserved`

Not used.

`u.DeviceSpecificData`

`u.DeviceSpecificData.DataSize`

Specifies the number of bytes appended to the end of the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure.

`u.DeviceSpecificData.Reserved1`

Not used.

`u.DeviceSpecificData.Reserved2`

Not used.

`u.Memory40`

`u.Memory40.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory40.Length40`

Contains the high 32 bits of the 40-bit length, in bytes, of the range of allocated memory addresses. The lowest 8 bits are treated as zero.

`u.Memory48`

`u.Memory48.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory48.Length48`

Contains the high 32 bits of the 48-bit length, in bytes, of the range of allocated memory addresses. The lowest 16 bits are treated as zero.

`u.Memory64`

`u.Memory64.Start`

For raw resources: Specifies the bus-relative physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For translated resources: Specifies the system physical address of the lowest of a range of contiguous memory addresses that are allocated to the device.

For more information about raw and translated resources, see Remarks.

`u.Memory64.Length64`

Contains the high 32 bits of the 64-bit length, in bytes, of the range of allocated memory addresses. The lowest 32 bits are treated as zero.

`u.Connection`

`u.Connection.Class`

Specifies the connection class. This member is set to one of the following values.

VALUE	MEANING
CM_RESOURCE_CONNECTION_CLASS_GPIO	Access the device through one or more pins on a GPIO controller.
CM_RESOURCE_CONNECTION_CLASS_SERIAL	Access the device through a serial bus or serial port.

`u.Connection.Type`

Specifies the connection type.

If **Class** = CM_RESOURCE_CONNECTION_CLASS_GPIO, **Type** is set to the following value.

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_GPIO_IO	Access the device through GPIO pins that are configured for I/O.

Note A GPIO pin that is configured as an interrupt request input is accessed as an ordinary interrupt resource (**CmResourceTypeInterrupt**).

If **Class** = CM_RESOURCE_CONNECTION_CLASS_SERIAL, **Type** is set to one of the following values.

VALUE	MEANING
CM_RESOURCE_CONNECTION_TYPE_SERIAL_I2C	The device is connected to an I^2C bus.
CM_RESOURCE_CONNECTION_TYPE_SERIAL_SPI	The device is connected to an SPI bus.

CM_RESOURCE_CONNECTION_TYPE_SERIAL_UART	The device is connected to a serial port.

`u.Connection.Reserved1`

Not used.

`u.Connection.Reserved2`

Not used.

`u.Connection.IdLowPart`

Contains the lower 32 bits of the 64-bit connection ID.

`u.Connection.IdHighPart`

Contains the upper 32 bits of the 64-bit connection ID.

Remarks

A **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure can describe either a raw (bus-relative) resource or a translated (system physical) resource, depending on the routine or IRP with which it is being used. For more information, see Raw and Translated Resources and IRP_MN_START_DEVICE.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FLOPPY_DEVICE_DATA

CM_FULL_RESOURCE_DESCRIPTOR

CM_KEYBOARD_DEVICE_DATA

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_SCSI_DEVICE_DATA

CM_SERIAL_DEVICE_DATA

DMA_OPERATIONS

IO_RESOURCE_DESCRIPTOR

IRP_MN_START_DEVICE

IoConnectInterrupt

IoGetDeviceProperty

IoReportResourceForDetection

KAFFINITY

RtlCmDecodeMemIoResource

RtlCmEncodeMemIoResource

CM_PARTIAL_RESOURCE_LIST structure

1/30/2019 • 2 minutes to read • Edit Online

The **CM_PARTIAL_RESOURCE_LIST** structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.

Syntax

```
typedef struct _CM_PARTIAL_RESOURCE_LIST {
  USHORT                          Version;
  USHORT                          Revision;
  ULONG                           Count;
  CM_PARTIAL_RESOURCE_DESCRIPTOR PartialDescriptors[1];
} CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;
```

Members

`Version`

The version number of this structure. This value should be 1.

`Revision`

The revision of this structure. This value should be 1.

`Count`

The number of elements contained in the **PartialDescriptors** array.

`PartialDescriptors`

The first element in an array of one or more CM_PARTIAL_RESOURCE_DESCRIPTOR structures.

Remarks

This structure is the header for an array of **CM_PARTIAL_RESOURCE_DESCRIPTOR** structures. The **PartialDescriptors** member contains the first element in this array, and the **Count** member specifies the total number of array elements. If the array contains more than one element, the remaining elements in the array immediately follow the **CM_PARTIAL_RESOURCE_LIST** structure in memory. The total number of bytes occupied by the **CM_PARTIAL_RESOURCE_LIST** structure and any array elements that follow this structure is **sizeof**(**CM_PARTIAL_RESOURCE_LIST**) + (**Count** - 1) * **sizeof**(**CM_PARTIAL_RESOURCE_DESCRIPTOR**).

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FULL_RESOURCE_DESCRIPTOR

CM_RESOURCE_LIST structure

1/30/2019 • 3 minutes to read • Edit Online

The **CM_RESOURCE_LIST** structure specifies all of the system hardware resources assigned to a device.

Syntax

```
typedef struct _CM_RESOURCE_LIST {
  ULONG                        Count;
  CM_FULL_RESOURCE_DESCRIPTOR List[1];
} CM_RESOURCE_LIST, *PCM_RESOURCE_LIST;
```

Members

`Count`

The number of full resource descriptors that are specified by this **CM_RESOURCE_LIST** structure. The **List** member is the header for the first full resource descriptor. For WDM drivers, **Count** is always 1.

`List`

The CM_FULL_RESOURCE_DESCRIPTOR structure that serves as the header for the first full resource descriptor. If the **CM_RESOURCE_LIST** structure contains more than one full resource descriptor, the second full resource descriptor immediately follows the first in memory, and so on. The size of each full resource descriptor depends on the length of the CM_PARTIAL_RESOURCE_DESCRIPTOR array that it contains. For more information, see the following Remarks section.

Remarks

This structure describes the assignment of hardware resources to a device. An IRP_MN_START_DEVICE IRP uses this structure to specify the resources that the Plug and Play manager assigns to a device. Drivers for legacy devices use this structure to pass their resource requirements to the IoReportResourceForDetection routine. For more information about hardware resource allocation, see Hardware Resources.

The **CM_RESOURCE_LIST** structure is a header for a larger data structure, of variable size, that contains one or more full resource descriptors. All of the data in this larger structure occupies a contiguous block of memory. Each full resource descriptor occupies a subblock within the larger block.

A full resource descriptor begins with a CM_FULL_RESOURCE_DESCRIPTOR structure, which serves as a header for an array of CM_PARTIAL_RESOURCE_DESCRIPTOR structures. The length of this array determines the size of the full resource descriptor. The last member in the **CM_FULL_RESOURCE_DESCRIPTOR** structure is a CM_PARTIAL_RESOURCE_LIST structure that contains, as its last member, the first element in this array. If the array contains more than one element, the remaining elements immediately follow, in memory, the end of the **CM_PARTIAL_RESOURCE_LIST** structure, which is also the end of the **CM_FULL_RESOURCE_DESCRIPTOR** structure.

Driver code can use pointer arithmetic to step from one full resource descriptor to the next. For example, if a parameter named *list* is a pointer to the **CM_FULL_RESOURCE_DESCRIPTOR** structure at the start of one full resource descriptor, *list* can be updated to point to the start of the next full resource descriptor as follows:

```
    list = (PCM_FULL_RESOURCE_DESCRIPTOR)(list->PartialResourceList.PartialDescriptors +
                                          list->PartialResourceList.Count);
```

In this example, `list->PartialResourceList.PartialDescriptors` is a pointer to the start of the **CM_PARTIAL_RESOURCE_DESCRIPTOR** array, and `list->PartialResourceList.Count` is the number of elements in the array. For more information about the **PartialDescriptors** and **Count** members, see CM_PARTIAL_RESOURCE_LIST. #### Examples All PnP drivers must handle IRP_MN_START_DEVICE IRPs. Typically, a driver's handler for this IRP walks the lists of assigned resources that are pointed to by the **Parameters.StartDevice.AllocatedResources** and **Parameters.StartDevice.AllocatedResourcesTranslated** members of the IO_STACK_LOCATION structure in the IRP. The following code example contains a function—named GetAssignedResources—that is called in the handler to walk each list. This function verifies that the required resources are specified in the list, and configures the device to use the resources. The GetAssignedResources function returns **TRUE** if it succeeds. Otherwise, it returns **FALSE** (probably from the **switch** statement, although the details are omitted to simplify the code example).

```
/* Process the assigned hardware resources. */

BOOLEAN GetAssignedResources(PCM_RESOURCE_LIST reslist)
{
    PCM_FULL_RESOURCE_DESCRIPTOR list;

    list = reslist->List;

    for (int ix = 0; ix < reslist->Count; ++ix)
    {
        /* Process resources in CM_FULL_RESOURCE_DESCRIPTOR block number ix. */

        for (int jx = 0; jx < list->PartialResourceList.Count; ++jx)
        {
            PCM_PARTIAL_RESOURCE_DESCRIPTOR desc;

            desc = list->PartialResourceList.PartialDescriptors + jx;

            switch (desc->Type)
            {
                /* Process element jx in PartialDescriptors array. */

                ...

            }
        }

        /* Advance to next CM_FULL_RESOURCE_DESCRIPTOR block in memory. */

        list = (PCM_FULL_RESOURCE_DESCRIPTOR)(list->PartialResourceList.PartialDescriptors +
                                              list->PartialResourceList.Count);
    }

    return TRUE;
}
```

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

CM_FLOPPY_DEVICE_DATA

CM_FULL_RESOURCE_DESCRIPTOR

CM_KEYBOARD_DEVICE_DATA

CM_PARTIAL_RESOURCE_LIST

CM_RESOURCE_LIST

CM_SCSI_DEVICE_DATA

CM_SERIAL_DEVICE_DATA

IRP_MN_START_DEVICE

IoConnectInterrupt

IoGetDeviceProperty

IoReportResourceForDetection

INTERFACE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **INTERFACE_TYPE** enumeration indicates the bus type.

Syntax

```
typedef enum _INTERFACE_TYPE {
  InterfaceTypeUndefined,
  Internal,
  Isa,
  Eisa,
  MicroChannel,
  TurboChannel,
  PCIBus,
  VMEBus,
  NuBus,
  PCMCIABus,
  CBus,
  MPIBus,
  MPSABus,
  ProcessorInternal,
  InternalPowerBus,
  PNPISABus,
  PNPBus,
  Vmcs,
  ACPIBus,
  MaximumInterfaceType
} INTERFACE_TYPE, *PINTERFACE_TYPE;
```

Constants

InterfaceTypeUndefined	Indicates that the interface type is undefined.
Internal	For internal use only.
Isa	Indicates that the interface is published by the ISA bus driver.
Eisa	Indicates that the interface is published by the EISA bus driver.
MicroChannel	Indicates that the interface is published by the MicroChannel bus driver.
TurboChannel	Indicates that the interface is published by the TurboChannel bus driver.
PCIBus	Indicates that the interface is published by the PCI bus driver.
VMEBus	Indicates that the interface is published by the VME bus driver.
NuBus	Indicates that the interface is published by the NuBus driver.

PCMCIABus	Indicates that the interface is published by the PCMCIA bus driver.
CBus	Indicates that the interface is published by the Cbus driver.
MPIBus	Indicates that the interface is published by the MPI bus driver.
MPSABus	Indicates that the interface is published by the MPSA bus driver.
ProcessorInternal	Indicates that the interface is published by the ISA bus driver.
InternalPowerBus	Indicates that the interface is published for an internal power bus. Some devices have power control ports that allow them to share power control with other devices. The Windows architecture represents these devices as slots on a virtual bus called an "internal power bus."
PNPISABus	Indicates that the interface is published by the PNPISA bus driver.
PNPBus	Indicates that the interface is published by the PNP bus driver.
Vmcs	Reserved for use by the operating system.
ACPIBus	Indicates that the interface is published by the ACPI bus driver. The ACPI bus driver enumerates devices that are described in the ACPI firmware of the hardware platform. These devices might physically reside on buses that are controlled by other bus drivers, but the ACPI bus driver must enumerate these devices because the other bus drivers cannot detect them. This interface type is defined starting with Windows 8.
MaximumInterfaceType	Marks the upper limit of the possible bus types.

Requirements

Header	wudfwdm.h (include Wdm.h, Miniport.h, Wudfwdm.h)

See Also

HW_INITIALIZATION_DATA

PORT_CONFIGURATION_INFORMATION

ScsiPortGetDeviceBase

ScsiPortValidateRange

StorPortGetDeviceBase

StorPortValidateRange

KINTERRUPT_MODE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **KINTERRUPT_MODE** enumeration type indicates whether an interrupt is level-triggered or edge-triggered.

Syntax

```
typedef enum _KINTERRUPT_MODE {
  LevelSensitive,
  Latched
} KINTERRUPT_MODE;
```

Constants

LevelSensitive	The interrupt is level-triggered. This is the mode for traditional PCI line-based interrupts.
Latched	The interrupt is edge-triggered. This is the mode for PCI message-signaled interrupts.

Remarks

The interrupt mode is an electrical characteristic of the interrupt. A device signals a level-triggered interrupt by changing the voltage on the interrupt pin, and holding it there until the processor signals that the interrupt is received. In contrast, to signal an edge-triggered interrupt, the device changes the voltage state for a fixed period of time, before reverting to the original voltage state.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

IoConnectInterrupt

IoConnectInterruptEx

KINTERRUPT_POLARITY

MEMORY_CACHING_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **MEMORY_CACHING_TYPE** enumeration type specifies the permitted caching behavior when allocating or mapping memory.

Syntax

```
typedef enum _MEMORY_CACHING_TYPE {
  MmNonCached,
  MmCached,
  MmWriteCombined,
  MmHardwareCoherentCached,
  MmNonCachedUnordered,
  MmUSWCCached,
  MmMaximumCacheType,
  MmNotMapped
} MEMORY_CACHING_TYPE;
```

Constants

MmNonCached	The requested memory should not be cached by the processor.
MmCached	The processor should cache the requested memory.
MmWriteCombined	The requested memory should not be cached by the processor, but writes to the memory can be combined by the processor.
MmHardwareCoherentCached	Reserved for system use.
MmNonCachedUnordered	Reserved for system use.
MmUSWCCached	Reserved for system use.
MmMaximumCacheType	Reserved for system use.
MmNotMapped	

Remarks

Processor translation buffers cache virtual to physical address translations. These translation buffers allow many virtual addresses to map a single physical address. However, only one caching behavior is allowed for any given physical address translation. Therefore, if a driver maps two different virtual address ranges to the same physical address, it must ensure that it specifies the same caching behavior for both. Otherwise, the processor behavior is undefined with unpredictable system results.

Requirements

Minimum supported client	Supported in Windows 2000 and later versions of Windows.
Header	wudfwdm.h (include Ntddk.h, Wdm.h, Miniport.h, Wudfwdm.h)

See Also

MmAllocateContiguousMemorySpecifyCache

MmFreeContiguousMemorySpecifyCache

MmMapIoSpace

MmMapLockedPagesSpecifyCache

MmMapLockedPagesWithReservedMapping

PINTERFACE_DEREFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceDereference* routine decrements the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_DEREFERENCE PinterfaceDereference;

void PinterfaceDereference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceReference routine to increment the reference count for the interface.

The driver that imports the interface is responsible for calling the *InterfaceDereference* routine to decrement the reference count after the driver is no longer using the interface. For example, a driver that requests a pointer to the interface by sending an IRP_MN_QUERY_INTERFACE request calls *InterfaceDereference*. Also, a driver that receives a pointer to the interface to another driver must call *InterfaceDereference*.

Requirements

Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceReference

PINTERFACE_REFERENCE callback function

1/25/2019 • 2 minutes to read • Edit Online

The *InterfaceReference* routine increments the reference count on a driver-defined interface.

Syntax

```
PINTERFACE_REFERENCE PinterfaceReference;

void PinterfaceReference(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to interface-specific context information. The caller passes the value that is passed as the **Context** member of the INTERFACE structure for the interface.

Return Value

None

Remarks

You can use the InterfaceDereference routine to decrement the reference count for the interface.

The driver that exports the interface is responsible for calling *InterfaceReference* to increment the reference count before the driver exports the interface. For example, the driver that initially exported the interface in response to an IRP_MN_QUERY_INTERFACE request calls *InterfaceReference*. Also, a driver that passes a pointer to the interface to another driver must call *InterfaceReference*.

Requirements

Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)

See Also

INTERFACE

IRP_MN_QUERY_INTERFACE

InterfaceDereference

DEVICE_REGISTRY_PROPERTY Enumeration

1/25/2019 • 2 minutes to read • Edit Online

The **DEVICE_REGISTRY_PROPERTY** enumeration identifies device properties that are stored in the registry.

Syntax

```
typedef enum  {
  DevicePropertyDeviceDescription,
  DevicePropertyHardwareID,
  DevicePropertyCompatibleIDs,
  DevicePropertyBootConfiguration,
  DevicePropertyBootConfigurationTranslated,
  DevicePropertyClassName,
  DevicePropertyClassGuid,
  DevicePropertyDriverKeyName,
  DevicePropertyManufacturer,
  DevicePropertyFriendlyName,
  DevicePropertyLocationInformation,
  DevicePropertyPhysicalDeviceObjectName,
  DevicePropertyBusTypeGuid,
  DevicePropertyLegacyBusType,
  DevicePropertyBusNumber,
  DevicePropertyEnumeratorName,
  DevicePropertyAddress,
  DevicePropertyUINumber,
  DevicePropertyInstallState,
  DevicePropertyRemovalPolicy,
  DevicePropertyResourceRequirements,
  DevicePropertyAllocatedResources,
  DevicePropertyContainerID
} DEVICE_REGISTRY_PROPERTY;
```

Constants

DevicePropertyDeviceDescription	
DevicePropertyHardwareID	
DevicePropertyCompatibleIDs	
DevicePropertyBootConfiguration	
DevicePropertyBootConfigurationTranslated	
DevicePropertyClassName	
DevicePropertyClassGuid	
DevicePropertyDriverKeyName	
DevicePropertyManufacturer	

DevicePropertyFriendlyName	
DevicePropertyLocationInformation	
DevicePropertyPhysicalDeviceObjectName	
DevicePropertyBusTypeGuid	
DevicePropertyLegacyBusType	
DevicePropertyBusNumber	
DevicePropertyEnumeratorName	
DevicePropertyAddress	
DevicePropertyUINumber	
DevicePropertyInstallState	
DevicePropertyRemovalPolicy	
DevicePropertyResourceRequirements	
DevicePropertyAllocatedResources	
DevicePropertyContainerID	

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

IoGetDeviceProperty

IO_STATUS_BLOCK structure

1/30/2019 • 2 minutes to read • Edit Online

A driver sets an IRP's I/O status block to indicate the final status of an I/O request, before calling IoCompleteRequest for the IRP.

Syntax

```
typedef struct _IO_STATUS_BLOCK {
  union {
    NTSTATUS Status;
    PVOID    Pointer;
  } DUMMYUNIONNAME;
  ULONG_PTR Information;
} *PIO_STATUS_BLOCK, IO_STATUS_BLOCK;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Status`

This is the completion status, either STATUS_SUCCESS if the requested operation was completed successfully or an informational, warning, or error STATUS_*XXX* value. For more information, see Using NTSTATUS values.

`DUMMYUNIONNAME.Pointer`

Reserved. For internal use only.

`Information`

This is set to a request-dependent value. For example, on successful completion of a transfer request, this is set to the number of bytes transferred. If a transfer request is completed with another STATUS_*XXX*, this member is set to zero.

Remarks

Unless a driver's dispatch routine completes an IRP with an error status value, the lowest-level driver in the chain frequently sets the IRP's I/O status block to the values that will be returned to the original requester of the I/O operation.

The IoCompletion routines of higher-level drivers usually check the I/O status block in IRPs completed by lower drivers. By design, the I/O status block in an IRP is the only information passed back from the underlying device driver to all higher-level drivers' *IoCompletion* routines.

The operating system implements support routines that write **IO_STATUS_BLOCK** values to caller-supplied output buffers. For example, see ZwOpenFile or NtOpenFile. These routines return status codes that might not match the status codes in the **IO_STATUS_BLOCK** structures. If one of these routines returns STATUS_PENDING, the caller should wait for the I/O operation to complete, and then check the status code in the **IO_STATUS_BLOCK** structure to determine the final status of the operation. If the routine returns a status code other than STATUS_PENDING, the caller should rely on this status code instead of the status code in the **IO_STATUS_BLOCK** structure.

For more information, see I/O Status Blocks.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

IO_STACK_LOCATION

IRP

IoCompleteRequest

IoSetCompletionRoutine

OBJECT_ATTRIBUTES structure

1/30/2019 • 4 minutes to read • Edit Online

The **OBJECT_ATTRIBUTES** structure specifies attributes that can be applied to objects or object handles by routines that create objects and/or return handles to objects.

Syntax

```
typedef struct _OBJECT_ATTRIBUTES {
  ULONG           Length;
  HANDLE          RootDirectory;
  PUNICODE_STRING ObjectName;
  ULONG           Attributes;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

Members

`Length`

The number of bytes of data contained in this structure. The InitializeObjectAttributes macro sets this member to **sizeof**(**OBJECT_ATTRIBUTES**).

`RootDirectory`

Optional handle to the root object directory for the path name specified by the **ObjectName** member. If **RootDirectory** is **NULL**, **ObjectName** must point to a fully qualified object name that includes the full path to the target object. If **RootDirectory** is non-**NULL**, **ObjectName** specifies an object name relative to the **RootDirectory** directory. The **RootDirectory** handle can refer to a file system directory or an object directory in the object manager namespace.

`ObjectName`

Pointer to a Unicode string that contains the name of the object for which a handle is to be opened. This must either be a fully qualified object name, or a relative path name to the directory specified by the **RootDirectory** member.

`Attributes`

Bitmask of flags that specify object handle attributes. This member can contain one or more of the flags in the following table.

FLAG	MEANING
OBJ_INHERIT	This handle can be inherited by child processes of the current process.

OBJ_PERMANENT	This flag only applies to objects that are named within the object manager. By default, such objects are deleted when all open handles to them are closed. If this flag is specified, the object is not deleted when all open handles are closed. Drivers can use the ZwMakeTemporaryObject routine to make a permanent object non-permanent.
OBJ_EXCLUSIVE	If this flag is set and the **OBJECT_ATTRIBUTES** structure is passed to a routine that creates an object, the object can be accessed exclusively. That is, once a process opens such a handle to the object, no other processes can open handles to this object. If this flag is set and the **OBJECT_ATTRIBUTES** structure is passed to a routine that creates an object handle, the caller is requesting exclusive access to the object for the process context that the handle was created in. This request can be granted only if the OBJ_EXCLUSIVE flag was set when the object was created.
OBJ_CASE_INSENSITIVE	If this flag is specified, a case-insensitive comparison is used when matching the name pointed to by the **ObjectName** member against the names of existing objects. Otherwise, object names are compared using the default system settings.
OBJ_OPENIF	If this flag is specified, by using the object handle, to a routine that creates objects and if that object already exists, the routine should open that object. Otherwise, the routine creating the object returns an NTSTATUS code of STATUS_OBJECT_NAME_COLLISION.
OBJ_OPENLINK	If an object handle, with this flag set, is passed to a routine that opens objects and if the object is a symbolic link object, the routine should open the symbolic link object itself, rather than the object that the symbolic link refers to (which is the default behavior).
OBJ_KERNEL_HANDLE	The handle is created in system process context and can only be accessed from kernel mode.
OBJ_FORCE_ACCESS_CHECK	The routine that opens the handle should enforce all access checks for the object, even if the handle is being opened in kernel mode.
OBJ_VALID_ATTRIBUTES	Reserved.

`SecurityDescriptor`

Specifies a security descriptor (SECURITY_DESCRIPTOR) for the object when the object is created. If this member is **NULL**, the object will receive default security settings.

`SecurityQualityOfService`

Optional quality of service to be applied to the object when it is created. Used to indicate the security impersonation level and context tracking mode (dynamic or static). Currently, the InitializeObjectAttributes macro sets this member to **NULL**.

Remarks

Use the InitializeObjectAttributes macro to initialize the members of the **OBJECT_ATTRIBUTES** structure. Note that **InitializeObjectAttributes** initializes the **SecurityQualityOfService** member to **NULL**. If you must specify

a non-**NULL** value, set the **SecurityQualityOfService** member after initialization.

To apply the attributes contained in this structure to an object or object handle, pass a pointer to this structure to a routine that accesses objects or returns object handles, such as ZwCreateFile or ZwCreateDirectoryObject.

All members of this structure are read-only. If a member of this structure is a pointer, the object that this member points to is read-only as well. Read-only members and objects can be used to acquire relevant information but must not be modified. To set the members of this structure, use the **InitializeObjectAttributes** macro.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE flag for the **Attributes** member (by using the **InitializeObjectAttributes** macro). This restricts the use of a handle opened for that object to processes running only in kernel mode. Otherwise, the handle can be accessed by the process in whose context the driver is running.

Requirements

Header	wudfwdm.h (include D3dkmthk.h, Ntdef.h, Wdm.h, Ntddk.h, Ntifs.h, Fltkernel.h)

See Also

FltCreateCommunicationPort

FltCreateFile

FltCreateFileEx

FltCreateFileEx2

InitializeObjectAttributes

IoCreateFile

IoCreateFileEx

IoCreateFileSpecifyDeviceObjectHint

ZwCreateDirectoryObject

ZwCreateFile

PO_FX_COMPONENT_IDLE_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_IDLE_STATE** structure specifies the attributes of an Fx power state of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_IDLE_STATE {
  ULONGLONG TransitionLatency;
  ULONGLONG ResidencyRequirement;
  ULONG     NominalPower;
} PO_FX_COMPONENT_IDLE_STATE, *PPO_FX_COMPONENT_IDLE_STATE;
```

Members

`TransitionLatency`

The transition latency. This latency is the amount of time, in 100-nanosecond units, that the component requires to return from this Fx state to the F0 state. For a **PO_FX_COMPONENT_IDLE_STATE** structure that specifies the attributes of the F0 state, set this member to zero.

`ResidencyRequirement`

The residency requirement. The residency requirement is the minimum amount of time, in 100-nanosecond units, that the component must spend in this Fx state to make a transition to this Fx state worthwhile. PoFx uses this member value as a hint to avoid switching a component to an Fx state unless the component is likely to remain in this state for at least the amount of time specified by **ResidencyRequirement**. For a **PO_FX_COMPONENT_IDLE_STATE** structure that describes the attributes of the F0 state, set this member to zero.

`NominalPower`

The power, in microwatts, that the component consumes in this Fx state. Set this member to **PO_FX_UNKNOWN_POWER** to indicate that PoFx should ignore (treat as negligible) the component's internal power consumption in this Fx state when PoFx evaluates which power state to switch to when the component is idle.

Remarks

The PO_FX_COMPONENT structure contains a pointer to an array of **PO_FX_COMPONENT_IDLE_STATE** structures. Each array element specifies the attributes of an Fx state. Element 0 describes F0, element 1 describes F1, and so on.

When the driver registers a device with PoFx, the driver supplies an array of **PO_FX_COMPONENT** structures. Each array element describes the power attributes of a component in the device.

Do not set **TransitionLatency** and **ResidencyRequirement** to **PO_FX_UNKNOWN_TIME** because it may prevent the power management framework (PoFx) from selecting the idle state.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT

PO_FX_COMPONENT_PERF_INFO structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_PERF_INFO** structure describes all the sets of performance states for a single component within a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_PERF_INFO {
  ULONG                    PerfStateSetsCount;
  PO_FX_COMPONENT_PERF_SET PerfStateSets[ANYSIZE_ARRAY];
} PO_FX_COMPONENT_PERF_INFO, *PPO_FX_COMPONENT_PERF_INFO;
```

Members

`PerfStateSetsCount`

The number of elements in the array **PerfStateSets** array.

`PerfStateSets`

This member is the first element in an array of one or more PO_FX_COMPONENT_PERF_SET elements. Each **PO_FX_COMPONENT_PERF_SET** represents one set of supported performance states for a component. If the array contains more than one element, the additional elements immediately follow the first element.

Remarks

When a device driver calls the PoFxRegisterComponentPerfStates routine to register a component for performance state support with the power management framework (PoFx), the driver can supply a **PO_FX_COMPONENT_PERF_INFO** structure that describes the sets of the performance states supported by the component. The **PerfStateSets** member of the **PO_FX_COMPONENT_PERF_INFO** structure contains an array of PO_FX_COMPONENT_PERF_SET structures that represent the supported performance states.

Most drivers are expected to define a single set of performance states per component. For example, a driver might define one set of performance states to control the clock frequency for a component. However, some drivers may need to define more than one performance state set to control multiple dimensions of performance states for a component. For example, a driver might define two sets of performance states to control the clock frequency and bus bandwidth.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Component-Level Performance State Management

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

PO_FX_COMPONENT_PERF_SET structure

2/5/2019 • 2 minutes to read • Edit Online

The **PO_FX_COMPONENT_PERF_SET** structure represents a set of performance states for a single component within a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_PERF_SET {
  UNICODE_STRING        Name;
  ULONGLONG             Flags;
  PO_FX_PERF_STATE_UNIT Unit;
  PO_FX_PERF_STATE_TYPE Type;
  union {
    struct {
      ULONG             Count;
      PPO_FX_PERF_STATE States;
    } Discrete;
    struct {
      ULONGLONG Minimum;
      ULONGLONG Maximum;
    } Range;
  };
} PO_FX_COMPONENT_PERF_SET, *PPO_FX_COMPONENT_PERF_SET;
```

Members

`Name`

An optional character string used to describe the component property controlled by the performance state set. For example, "Clock frequency" or "Memory bandwidth".

`Flags`

Set to 0. Currently, no flags are defined for this member.

`Unit`

A PO_FX_PERF_STATE_UNIT value that specifies the type of unit that is controlled by the performance state set.

`Type`

A PO_FX_PERF_STATE_TYPE value that specifies the type of performance states in the set (a discrete number of states or a continuous distribution of states between a minimum and maximum value).

`Discrete`

`Discrete.Count`

The number of performance states in the set.

`Discrete.States`

A pointer to a PO_FX_PERF_STATE array. The length of this array is specified by the **Count** member. Each array element describes one power state in the set.

`Range`

`Range.Minimum`

The minimum performance state value.

`Range.Maximum`

The maximum performance state value.

Remarks

When a device driver calls the PoFxRegisterComponentPerfStates routine to register a component for performance state support with the power management framework (PoFx), the driver can supply a PO_FX_COMPONENT_PERF_INFO structure that describes the sets of the performance states supported by the component. The **PerfStateSets** member of the **PO_FX_COMPONENT_PERF_INFO** structure contains an array of **PO_FX_COMPONENT_PERF_SET** structures that represent the supported performance states.

Most drivers are expected to define a single set of performance states per component. For example, a driver might define one set of performance states to control the clock frequency for a component. However, some drivers may need to define more than one performance state set to control multiple dimensions of performance states for a component. For example, a driver might define two sets of performance states to control the clock frequency and bus bandwidth.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PO_FX_COMPONENT_PERF_INFO

PoFxRegisterComponentPerfStates

PO_FX_COMPONENT_V1 structure

1/30/2019 • 3 minutes to read • Edit Online

The **PO_FX_COMPONENT** structure describes the power state attributes of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_V1 {
  GUID                          Id;
  ULONG                         IdleStateCount;
  ULONG                         DeepestWakeableIdleState;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
} PO_FX_COMPONENT_V1, *PPO_FX_COMPONENT_V1;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The driver should specify a nonzero value for this member if the power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of Fx power states that the component supports. A component must support at least one Fx state (F0).

`DeepestWakeableIdleState`

The index of the deepest Fx state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStates`

A pointer to a PO_FX_COMPONENT_IDLE_STATE array. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an Fx power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

Remarks

When a device driver registers a device with PoFx, the driver supplies a PO_FX_DEVICE structure that holds the registration information. This structure contains an array of **PO_FX_COMPONENT** structures. The elements in this array describe the power attributes of the individual components in the device. PoFx uses the information in this array to manage the power settings of these components.

The **Id** member contains a component ID that PoFx can use to uniquely identify a component. Do not confuse the component ID with the component index, which a routine such as PoFxActivateComponent uses to identify a component in a registered device. A component index is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes are integer values in the range 0 to N–1. In contrast, a component ID is a

GUID value.

The **Flags** member may be set to the **PO_FX_COMPONENT_FLAG_F0_ON_DX** value if the driver specified support for **PO_FX_VERSION_V2** during registration with PoFx (that is, during the call to the PoFxRegisterDevice routine, the driver specified the value **PO_FX_VERSION_V2** for the **Version** member of the PO_FX_DEVICE structure that was passed to the *Device* parameter). If the **PO_FX_COMPONENT_FLAG_F0_ON_DX** flag is specified, PoFx will place the component into the F0 power state during Dx transitions and for wait/wake IRPs. Before dispatching a Dx IRP or wait/wake IRP to the device stack (whichever event happens first), the power manager will transition the component into the F0 state. The component will be kept in F0 until the wake/wait IRP is canceled or completed, or the Dx IRP is completed and driver has called the PoFxReportDevicePoweredOn routine (whichever event happens last).

Component dependencies

PoFx supports logical component dependencies within a device (but not across different devices). During device registration for each component, the driver can supply a number (the *ProviderCount* member) and an array of component indexes (the *Providers* member) this component depends on; these are known as *providers*. PoFx guarantees that the component will not become active until all of its providers are active.

Component dependencies form a directed graph where an edge points from a dependent component to a provider and represents a dependency. During device registration, PoFx validates the dependency graph: cycles and repeated edges are not allowed and the depth of the graph for the depth-first search traversal cannot exceed four. PoFx also sets the components' active reference count according to the dependency graph; that is, each provider's active reference count is incremented once for each component that depends on it.

During the component activation, PoFx asynchronously activates all providers first. After a component is active, PoFx checks every dependent component and continues their activation if necessary.

When the component transitions to the idle state, PoFx first makes the dependent components idle. After the driver completes the ComponentIdleConditionCallback and the platform extension plug-in (PEP) is notified that the component is idle, PoFx recursively traverses the dependency graph in breadth-first traversal and asynchronously idles each provider.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Component-Level Power Management

PO_FX_COMPONENT_IDLE_STATE

PO_FX_DEVICE

PO_FX_COMPONENT_V2 structure

1/30/2019 • 4 minutes to read • Edit Online

The **PO_FX_COMPONENT** structure describes the power state attributes of a component in a device.

Syntax

```
typedef struct _PO_FX_COMPONENT_V2 {
  GUID                          Id;
  ULONGLONG                     Flags;
  ULONG                         DeepestWakeableIdleState;
  ULONG                         IdleStateCount;
  PPO_FX_COMPONENT_IDLE_STATE IdleStates;
  ULONG                         ProviderCount;
  PULONG                        Providers;
} PO_FX_COMPONENT_V2, *PPO_FX_COMPONENT_V2;
```

Members

`Id`

A component ID that uniquely identifies this component with respect to the other components in the device. The driver should specify a nonzero value for this member if the power management framework (PoFx) requires a component ID to distinguish this component from other, similar components in the same device. This member is optional. If this member is not used, it must be set to all zeros.

`Flags`

This member can be set to the following flag bit: **PO_FX_COMPONENT_FLAG_F0_ON_DX**. For more information, see Remarks.

`DeepestWakeableIdleState`

The index of the deepest Fx state from which the component can wake. Specify 0 for F0, 1 for F1, and so on. This index must be less than **IdleStateCount**.

`IdleStateCount`

The number of elements in the array that is pointed to by the **IdleStates** member. Additionally, this member specifies the number of Fx power states that the component supports. A component must support at least one Fx state (F0).

`IdleStates`

A pointer to a PO_FX_COMPONENT_IDLE_STATE array. The length of this array is specified by the **IdleStateCount** member. Each array element specifies the attributes of an Fx power state that is supported by the component. Element 0 describes F0, element 1 describes F1, and so on.

`ProviderCount`

A value that specifies the number of components this component depends on. For more information, see *Component dependencies* under Remarks.

`Providers`

A pointer to an array of component indexes this component depends on. For more information, see *Component dependencies* under Remarks.

Remarks

When a device driver registers a device with PoFx, the driver supplies a PO_FX_DEVICE structure that holds the registration information. This structure contains an array of **PO_FX_COMPONENT** structures. The elements in this array describe the power attributes of the individual components in the device. PoFx uses the information in this array to manage the power settings of these components.

The **Id** member contains a component ID that PoFx can use to uniquely identify a component. Do not confuse the component ID with the component index, which a routine such as PoFxActivateComponent uses to identify a component in a registered device. A component index is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes are integer values in the range 0 to N–1. In contrast, a component ID is a GUID value.

The **Flags** member may be set to the **PO_FX_COMPONENT_FLAG_F0_ON_DX** value if the driver specified support for **PO_FX_VERSION_V2** during registration with PoFx (that is, during the call to the PoFxRegisterDevice routine, the driver specified the value **PO_FX_VERSION_V2** for the **Version** member of the PO_FX_DEVICE structure that was passed to the *Device* parameter). If the **PO_FX_COMPONENT_FLAG_F0_ON_DX** flag is specified, PoFx will place the component into the F0 power state during Dx transitions and for wait/wake IRPs. Before dispatching a Dx IRP or wait/wake IRP to the device stack (whichever event happens first), the power manager will transition the component into the F0 state. The component will be kept in F0 until the wake/wait IRP is canceled or completed, or the Dx IRP is completed and driver has called the PoFxReportDevicePoweredOn routine (whichever event happens last).

Component dependencies

PoFx supports logical component dependencies within a device (but not across different devices). During device registration for each component, the driver can supply a number (the *ProviderCount* member) and an array of component indexes (the *Providers* member) this component depends on; these are known as *providers*. PoFx guarantees that the component will not become active until all of its providers are active.

Component dependencies form a directed graph where an edge points from a dependent component to a provider and represents a dependency. During device registration, PoFx validates the dependency graph: cycles and repeated edges are not allowed and the depth of the graph for the depth-first search traversal cannot exceed four. PoFx also sets the components' active reference count according to the dependency graph; that is, each provider's active reference count is incremented once for each component that depends on it.

During the component activation, PoFx asynchronously activates all providers first. After a component is active, PoFx checks every dependent component and continues their activation if necessary.

When the component transitions to the idle state, PoFx first makes the dependent components idle. After the driver completes the ComponentIdleConditionCallback and the platform extension plug-in (PEP) is notified that the component is idle, PoFx recursively traverses the dependency graph in breadth-first traversal and asynchronously idles each provider.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Component-Level Power Management

PO_FX_COMPONENT_IDLE_STATE

PO_FX_DEVICE

PO_FX_DEVICE_V1 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_DEVICE** structure describes the power attributes of a device to the power management framework (PoFx).

Syntax

```
typedef struct _PO_FX_DEVICE_V1 {
  ULONG                                    Version;
  ULONG                                    ComponentCount;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK   ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK       ComponentIdleStateCallback;
  PPO_FX_DEVICE_POWER_REQUIRED_CALLBACK      DevicePowerRequiredCallback;
  PPO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK  DevicePowerNotRequiredCallback;
  PPO_FX_POWER_CONTROL_CALLBACK              PowerControlCallback;
  PVOID                                    DeviceContext;
  PO_FX_COMPONENT_V1                       Components[ANYSIZE_ARRAY];
} PO_FX_DEVICE_V1, *PPO_FX_DEVICE_V1;
```

Members

`Version`

The version number of this structure. If the driver will register for performance state support with PoFx, set this member to **PO_FX_VERSION_V2**. Otherwise, set this member to **PO_FX_VERSION_V1**.

`ComponentCount`

The number of elements in the **Components** array. Additionally, this member specifies the number of components in the device.

`ComponentActiveConditionCallback`

A pointer to the ComponentActiveConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleConditionCallback`

A pointer to the ComponentIdleConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleStateCallback`

A pointer to the ComponentIdleStateCallback callback routine that is implemented by the device driver.

`DevicePowerRequiredCallback`

A pointer to the DevicePowerRequiredCallback callback routine that is implemented by the device driver.

`DevicePowerNotRequiredCallback`

A pointer to the DevicePowerNotRequiredCallback callback routine that is implemented by the device driver.

`PowerControlCallback`

A pointer to the PowerControlCallback callback routine that is implemented by the device driver.

`DeviceContext`

A pointer to a caller-allocated device context. This pointer is passed as a parameter to each driver-implemented callback function that is pointed to by this structure. The device driver uses this context to store information about the current power state of the device. This context is opaque to PoFx.

`Components`

This member is the first element in an array of one or more PO_FX_COMPONENT elements. If the array contains more than one element, the additional elements immediately follow the **PO_FX_DEVICE** structure. The array contains one element for each component in the device. The Fx power state of each component can be controlled independently of the Fx power states of the other components in the device. The **ANYSIZE_ARRAY** constant is defined to be 1 in the Ntdef.h header file.

Remarks

To register a device with PoFx, a driver calls the PoFxRegisterDevice routine and supplies, as a parameter, a pointer to a **PO_FX_DEVICE** structure that describes the device.

Each element in the **Components** array describes the power state attributes of one component in the device. Each component in the device is identified by its **Components** array index. Routines such as PoFxActivateComponent and PoFxCompleteIdleCondition use the array index of a component to identify the component.

A device driver is not required to implement all six callback routines. The driver can set a function pointer in the **PO_FX_DEVICE** structure to NULL if the driver does not implement the corresponding callback routine. However, certain callback routines must be implemented. Specifically, if one or more components in a device has more than one Fx state, the driver must implement ComponentIdleStateCallback, ComponentActiveConditionCallback, and ComponentIdleConditionCallback routines. Otherwise, device registration fails and PoFxRegisterDevice returns **STATUS_INVALID_PARAMETER**.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

ComponentActiveConditionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

DevicePowerNotRequiredCallback

DevicePowerRequiredCallback

PO_FX_COMPONENT

PoFxActivateComponent

PoFxCompleteIdleCondition

PoFxRegisterDevice

PowerControlCallback

PO_FX_DEVICE_V2 structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_DEVICE** structure describes the power attributes of a device to the power management framework (PoFx).

Syntax

```
typedef struct _PO_FX_DEVICE_V2 {
  ULONG                                    Version;
  ULONGLONG                                Flags;
  PPO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK ComponentActiveConditionCallback;
  PPO_FX_COMPONENT_IDLE_CONDITION_CALLBACK   ComponentIdleConditionCallback;
  PPO_FX_COMPONENT_IDLE_STATE_CALLBACK       ComponentIdleStateCallback;
  PPO_FX_DEVICE_POWER_REQUIRED_CALLBACK      DevicePowerRequiredCallback;
  PPO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK  DevicePowerNotRequiredCallback;
  PPO_FX_POWER_CONTROL_CALLBACK              PowerControlCallback;
  PVOID                                    DeviceContext;
  ULONG                                    ComponentCount;
  PO_FX_COMPONENT_V2                       Components[ANYSIZE_ARRAY];
} PO_FX_DEVICE_V2, *PPO_FX_DEVICE_V2;
```

Members

`Version`

The version number of this structure. If the driver will register for performance state support with PoFx, set this member to **PO_FX_VERSION_V2**. Otherwise, set this member to **PO_FX_VERSION_V1**.

`Flags`

Set to 0. Currently, no flags are defined for this member.

`ComponentActiveConditionCallback`

A pointer to the ComponentActiveConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleConditionCallback`

A pointer to the ComponentIdleConditionCallback callback routine that is implemented by the device driver.

`ComponentIdleStateCallback`

A pointer to the ComponentIdleStateCallback callback routine that is implemented by the device driver.

`DevicePowerRequiredCallback`

A pointer to the DevicePowerRequiredCallback callback routine that is implemented by the device driver.

`DevicePowerNotRequiredCallback`

A pointer to the DevicePowerNotRequiredCallback callback routine that is implemented by the device driver.

`PowerControlCallback`

A pointer to the PowerControlCallback callback routine that is implemented by the device driver.

`DeviceContext`

A pointer to a caller-allocated device context. This pointer is passed as a parameter to each driver-implemented callback function that is pointed to by this structure. The device driver uses this context to store information about the current power state of the device. This context is opaque to PoFx.

`ComponentCount`

The number of elements in the **Components** array. Additionally, this member specifies the number of components in the device.

`Components`

This member is the first element in an array of one or more PO_FX_COMPONENT elements. If the array contains more than one element, the additional elements immediately follow the **PO_FX_DEVICE** structure. The array contains one element for each component in the device. The Fx power state of each component can be controlled independently of the Fx power states of the other components in the device. The **ANYSIZE_ARRAY** constant is defined to be 1 in the Ntdef.h header file.

Remarks

To register a device with PoFx, a driver calls the PoFxRegisterDevice routine and supplies, as a parameter, a pointer to a **PO_FX_DEVICE** structure that describes the device.

Each element in the **Components** array describes the power state attributes of one component in the device. Each component in the device is identified by its **Components** array index. Routines such as PoFxActivateComponent and PoFxCompleteIdleCondition use the array index of a component to identify the component.

A device driver is not required to implement all six callback routines. The driver can set a function pointer in the **PO_FX_DEVICE** structure to NULL if the driver does not implement the corresponding callback routine. However, certain callback routines must be implemented. Specifically, if one or more components in a device has more than one Fx state, the driver must implement ComponentIdleStateCallback, ComponentActiveConditionCallback, and ComponentIdleConditionCallback routines. Otherwise, device registration fails and PoFxRegisterDevice returns **STATUS_INVALID_PARAMETER**.

Requirements

Minimum supported client	Supported starting with Windows 8.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

ComponentActiveConditionCallback

ComponentIdleConditionCallback

ComponentIdleStateCallback

DevicePowerNotRequiredCallback

DevicePowerRequiredCallback

PO_FX_COMPONENT

PoFxActivateComponent

PoFxCompleteIdleCondition

PoFxRegisterDevice

PowerControlCallback

PO_FX_PERF_STATE structure

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE** structure represents a performance state for a single component within a device.

Syntax

```
typedef struct _PO_FX_PERF_STATE {
  ULONGLONG Value;
  PVOID     Context;
} PO_FX_PERF_STATE, *PPO_FX_PERF_STATE;
```

Members

`Value`

The value of this performance state. The units are specified by the **Unit** member of the PO_FX_COMPONENT_PERF_SET that contains this performance state. For example, if **Value** is 100000000 and the **Unit** member of the **PO_FX_COMPONENT_PERF_SET** is **PoFxPerfStateUnitFrequency**, this performance state represents 100 MHz.

`Context`

A pointer to additional context for the performance state that cannot be presented by the **Value** member. This data can be shared between the driver and the platform extension plug-in (PEP).

> **Note** Using the **Context** member can hinder the visualization of performance states in Windows Performance Analyzer.

Remarks

The **States** member of the PO_FX_COMPONENT_PERF_SET structure contains an array of **PO_FX_PERF_STATE** elements.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PO_FX_COMPONENT_PERF_SET

PO_FX_PERF_STATE_UNIT

PoFxRegisterComponentPerfStates

PO_FX_PERF_STATE_CHANGE structure

2/5/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_CHANGE** structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.

Syntax

```
typedef struct _PO_FX_PERF_STATE_CHANGE {
  ULONG Set;
  union {
    ULONG     StateIndex;
    ULONGLONG StateValue;
  };
} PO_FX_PERF_STATE_CHANGE, *PPO_FX_PERF_STATE_CHANGE;
```

Members

`Set`

The index of the performance state set that is being changed within the collection of performance state sets for the component.

`StateIndex`

For sets that represent a discrete number of performance states (that is, where the **Type** member of the PO_FX_COMPONENT_PERF_SET is **PoFxPerfStateTypeDiscrete**), this member is the index of the performance state to use for the new performance level.

`StateValue`

For sets that represent a continuous distribution of performance states (that is, where the **Type** member of the PO_FX_COMPONENT_PERF_SET is **PoFxPerfStateTypeRange**), this member is the value of the performance state to use for the new performance level.

Remarks

The **PO_FX_PERF_STATE_CHANGE** structure is used for the *PerfChange* parameter of the PoFxIssueComponentPerfStateChange routine and the *PerfChanges* parameter of the PoFxIssueComponentPerfStateChangeMultiple routine.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

Device Performance State Management

PoFxIssueComponentPerfStateChange

PoFxIssueComponentPerfStateChangeMultiple

PO_FX_PERF_STATE_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_TYPE** enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.

Syntax

```
typedef enum _PO_FX_PERF_STATE_TYPE {
  PoFxPerfStateTypeDiscrete,
  PoFxPerfStateTypeRange,
  PoFxPerfStateTypeMaximum
} PO_FX_PERF_STATE_TYPE, *PPO_FX_PERF_STATE_TYPE;
```

Constants

PoFxPerfStateTypeDiscrete	Indicates that the performance state set contains a discrete number of states.
PoFxPerfStateTypeRange	Indicates that the performance state set contains a continuous distribution of states between a minimum and maximum value.
PoFxPerfStateTypeMaximum	This value is reserved for system use.

Remarks

The **Type** member of the PO_FX_COMPONENT_PERF_SET structure is a value from the **PO_FX_PERF_STATE_TYPE** enumeration.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

PO_FX_PERF_STATE_UNIT Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **PO_FX_PERF_STATE_UNIT** enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.

Syntax

```
typedef enum _PO_FX_PERF_STATE_UNIT {
  PoFxPerfStateUnitOther,
  PoFxPerfStateUnitFrequency,
  PoFxPerfStateUnitBandwidth,
  PoFxPerfStateUnitMaximum
} PO_FX_PERF_STATE_UNIT, *PPO_FX_PERF_STATE_UNIT;
```

Constants

PoFxPerfStateUnitOther	Indicates that the performance states control a unit other than frequency or bandwidth.
PoFxPerfStateUnitFrequency	Indicates that the performance states control frequency, measured in Hz.
PoFxPerfStateUnitBandwidth	Indicates that the performance states control bandwidth, measured in bits per second.
PoFxPerfStateUnitMaximum	This value is reserved for system use.

Remarks

The **Unit** member of the PO_FX_COMPONENT_PERF_SET structure is a value from the **PO_FX_PERF_STATE_UNIT** enumeration.

Requirements

Minimum supported client	Supported starting with Windows 10.
Header	wudfwdm.h (include Wudfwdm.h)

See Also

PO_FX_COMPONENT_PERF_SET

PoFxRegisterComponentPerfStates

POOL_TYPE Enumeration

1/30/2019 • 2 minutes to read • Edit Online

The **POOL_TYPE** enumeration type specifies the type of system memory to allocate.

Syntax

```
typedef enum _POOL_TYPE {
  NonPagedPool,
  NonPagedPoolExecute,
  PagedPool,
  NonPagedPoolMustSucceed,
  DontUseThisType,
  NonPagedPoolCacheAligned,
  PagedPoolCacheAligned,
  NonPagedPoolCacheAlignedMustS,
  MaxPoolType,
  NonPagedPoolBase,
  NonPagedPoolBaseMustSucceed,
  NonPagedPoolBaseCacheAligned,
  NonPagedPoolBaseCacheAlignedMustS,
  NonPagedPoolSession,
  PagedPoolSession,
  NonPagedPoolMustSucceedSession,
  DontUseThisTypeSession,
  NonPagedPoolCacheAlignedSession,
  PagedPoolCacheAlignedSession,
  NonPagedPoolCacheAlignedMustSSession,
  NonPagedPoolNx,
  NonPagedPoolNxCacheAligned,
  NonPagedPoolSessionNx
} POOL_TYPE;
```

Constants

NonPagedPool	*Nonpaged pool*, which is nonpageable system memory. Nonpaged pool can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary. System memory allocated with the **NonPagedPool** pool type is executable. For more information, see the description of the **NonPagedPoolExecute** pool type. Starting with Windows 8, drivers should allocate most or all of their nonpaged memory from the no-execute (NX) nonpaged pool instead of the executable nonpaged pool. For more information, see the description of the **NonPagedPoolNx** pool type.

NonPagedPoolExecute	Starting with Windows 8, **NonPagedPoolExecute** is an alternate name for the **NonPagedPool** value. This value indicates that the allocated memory is to be nonpaged and executable—that is, instruction execution is enabled in this memory. To port a driver from an earlier version of Windows, you should typically replace all or most instances of the **NonPagedPool** name in the driver source code with **NonPagedPoolNx**. Avoid replacing instances of the **NonPagedPool** name with **NonPagedPoolExecute** except in cases in which executable memory is explicitly required. For more information, see No-Execute (NX) Nonpaged Pool.
PagedPool	*Paged pool*, which is pageable system memory. Paged pool can only be allocated and accessed at IRQL < DISPATCH_LEVEL.
NonPagedPoolMustSucceed	This value is for internal use only, and is allowed only during system startup. Drivers must not specify this value at times other than system startup, because a "must succeed" request crashes the system if the requested memory size is unavailable.
DontUseThisType	Reserved for system use.
NonPagedPoolCacheAligned	Nonpaged pool, aligned on processor cache boundaries. This value is for internal use only.
PagedPoolCacheAligned	Paged pool, aligned on processor cache boundaries. This value is for internal use only.
NonPagedPoolCacheAlignedMustS	This value is for internal use only, and is allowed only during system startup. It is the cache-aligned equivalent of **NonPagedPoolMustSucceed**.
MaxPoolType	Reserved for system use.
NonPagedPoolBase	Reserved for system use.
NonPagedPoolBaseMustSucceed	Reserved for system use.
NonPagedPoolBaseCacheAligned	Reserved for system use.
NonPagedPoolBaseCacheAlignedMustS	Reserved for system use.
NonPagedPoolSession	Deprecated. Do not use.
PagedPoolSession	Deprecated. Do not use.
NonPagedPoolMustSucceedSession	Deprecated. Do not use.
DontUseThisTypeSession	Deprecated. Do not use.
NonPagedPoolCacheAlignedSession	Deprecated. Do not use.
PagedPoolCacheAlignedSession	Deprecated. Do not use.

NonPagedPoolCacheAlignedMustSSession	Deprecated. Do not use.
NonPagedPoolNx	*No-execute* (NX) nonpaged pool. This pool type is available starting with Windows 8. In contrast to the nonpaged pool designated by **NonPagedPool**, which allocates executable memory, the NX nonpaged pool allocates memory in which instruction execution is disabled. For more information, see No-Execute (NX) Nonpaged Pool. Nonpaged pool can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary.
NonPagedPoolNxCacheAligned	NX nonpaged pool, aligned on processor cache boundaries. This value is reserved for exclusive use by the operating system.
NonPagedPoolSessionNx	Reserved for exclusive use by the operating system.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)

See Also

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithTag

ExAllocatePoolWithTagPriority

ExInitializeLookasideListEx

ExInitializeNPagedLookasideList

ExInitializePagedLookasideList

LookasideListAllocateEx

SeAssignSecurity

SeAssignSecurityEx

UNICODE_STRING structure

1/30/2019 • 2 minutes to read • Edit Online

The **UNICODE_STRING** structure is used to define Unicode strings.

Syntax

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
#if ...
  USHORT *Buffer;
#else
  PWCH   Buffer;
#endif
} UNICODE_STRING;
```

Members

`Length`

The length, in bytes, of the string stored in **Buffer**.

`MaximumLength`

The length, in bytes, of **Buffer**.

`Buffer.#4025480185`

`Buffer.#407591603`

Remarks

The **UNICODE_STRING** structure is used to pass Unicode strings. Use RtlUnicodeStringInit or RtlUnicodeStringInitEx to initialize a **UNICODE_STRING** structure.

If the string is null-terminated, **Length** does not include the trailing null character.

The **MaximumLength** is used to indicate the length of **Buffer** so that if the string is passed to a conversion routine such as RtlAnsiStringToUnicodeString the returned string does not exceed the buffer size.

Requirements

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntdef.h)

See Also

ANSI_STRING

OEM_STRING

RtlAnsiStringToUnicodeSize

RtlAnsiStringToUnicodeString

RtlFreeUnicodeString

RtlInitUnicodeString

RtlUnicodeStringToAnsiSize

RtlUnicodeStringToAnsiString

AppendTailList function

1/25/2019 • 3 minutes to read • Edit Online

The **AppendTailList** routine appends a doubly linked list of LIST_ENTRY structures to the tail of another doubly linked list of **LIST_ENTRY** structures.

Syntax

```
void AppendTailList(
  PLIST_ENTRY ListHead,
  PLIST_ENTRY ListToAppend
);
```

Parameters

`ListHead`

A pointer to the head of the list to which the list pointed to by *ListToAppend* is to be appended. This parameter must be a valid, non-NULL pointer to the LIST_ENTRY structure that is the head of a list. This list can be empty.

`ListToAppend`

A pointer to the first entry in the list to append to the list pointed to by *ListHead*. This parameter must be a valid, non-NULL pointer to the LIST_ENTRY structure that is the first entry in a headless list. For more information, see Remarks.

Return Value

None.

Remarks

This routine constructs a list that contains all the combined entries from the input *ListHead* and *ListToAppend* lists. When the routine returns, the LIST_ENTRY structure pointed to by the *ListHead* parameter is the head of the resulting combined list. The **LIST_ENTRY** structure pointed to by the *ListToAppend* parameter is an entry in the resulting combined list.

The caller sets the *ListToAppend* parameter to point to the first entry in the list to be appended. In this regard, *AppendTailList* does not conform to the convention followed by the other doubly linked list routines that use the **LIST_ENTRY** structure. In these other routines, a parameter that represents a list points to the list head instead of to the first entry in the list. If your driver code uses a list head to represent the list to be appended, the list head must be removed from this list before the list is passed as the *ListToAppend* parameter to **AppendTailList**. For a code example that shows how to remove this list head, see Examples.

AppendTailList can be used to insert a single entry at the tail of a list, but the **Flink** and **Blink** members of the **LIST_ENTRY** structure pointed to by *ListToAppend* must first be initialized—typically, by calling the InitializeListHead routine. In contrast, the InsertTailList routine does not require the **Flink** and **Blink** members of the **LIST_ENTRY** structure pointed to by the *Entry* parameter to be initialized before the **InsertTailList** call. For this reason, **InsertTailList** might be more convenient to use to insert a single entry at the tail of a list.

A driver can access the **Flink** and **Blink** members of a **LIST_ENTRY**, but the members must only be updated by

the system routines supplied for this purpose.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **AppendTailList** can be running at any IRQL. If **AppendTailList** is called at IRQL >= DISPATCH_LEVEL, the storage for the list entries must be memory-resident.

Examples

The following code example shows how to write a function named `MyAppendTailList` that is similar to **AppendTailList**, but that treats the *ListToAppend* parameter as a pointer to a list head instead of as a pointer to the first entry in a (headless) list. Unlike **AppendTailList**, the `MyAppendTailList` function avoids including the **LIST_ENTRY** structure pointed to by the *ListToAppend* parameter in the entries that are appended to the list pointed to by the *ListHead* parameter.

```
VOID MyAppendTailList(
    _Inout_  PLIST_ENTRY ListHead,
    _Inout_  PLIST_ENTRY ListToAppend
    )
{
    PLIST_ENTRY entry = ListToAppend->Flink;

    if (!IsListEmpty(ListToAppend))
    {
        RemoveEntryList(ListToAppend);
        InitializeListHead(ListToAppend);
        AppendTailList(ListHead, entry);
    }
}
```

The `MyAppendTailList` function in this code example treats both the *ListHead* and *ListToAppend* parameters as pointers to list heads. When this function returns, the list pointed to by *ListToAppend* is empty; that is, it consists of a list head that has no associated list entries. All of the entries that were initially in this list have been appended to the list pointed to by *ListHead*.

Requirements

Minimum supported client	Available starting with Windows Vista.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

InitializeListHead

InsertTailList

LIST_ENTRY

RemoveEntryList

ARGUMENT_PRESENT macro

1/25/2019 • 2 minutes to read • Edit Online

The **ARGUMENT_PRESENT** macro evaluates to **FALSE** if the value of *ArgumentPointer* is **NULL**. Otherwise, it evaluates to **TRUE**.

Syntax

```
void ARGUMENT_PRESENT(
   ArgumentPointer
);
```

Parameters

`ArgumentPointer`

Specifies the value of the pointer argument to be tested.

Return Value

ARGUMENT_PRESENT returns FALSE if the value of ArgumentPointer is NULL. Otherwise, it returns TRUE.

Remarks

This macro can be called in conditional code to determine whether an optional argument has been passed in a call.

Requirements

Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntdef.h)

DRIVER_INITIALIZE callback function

1/25/2019 • 2 minutes to read • Edit Online

DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver.

Syntax

```
DRIVER_INITIALIZE DriverInitialize;

NTSTATUS DriverInitialize(
  _DRIVER_OBJECT *DriverObject,
  PUNICODE_STRING RegistryPath
)
{...}
```

Parameters

`*DriverObject`

A pointer to a DRIVER_OBJECT structure. This is the driver's driver object.

`RegistryPath`

A pointer to a counted Unicode string specifying the path to the driver's registry key.

Return Value

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in Ntstatus.h.

Remarks

The *DriverObject* parameter supplies the **DriverEntry** routine with a pointer to the driver's driver object, which is allocated by the I/O manager. The **DriverEntry** routine must fill in the driver object with entry points for the driver's standard routines.

The *DriverObject* pointer gives the driver access to *DriverObject*->**HardwareDatabase**, which points to a counted Unicode string that specifies a path to the registry's **\Registry\Machine\Hardware** tree.

The registry path string pointed to by *RegistryPath* is of the form **\Registry\Machine\System\CurrentControlSet\Services<i>DriverName**. A driver can use this path to store driver-specific information; see Registry Keys for Drivers. The **DriverEntry** routine should save a copy of the Unicode string, not the pointer, since the I/O manager frees the *RegistryPath* buffer after **DriverEntry** returns.

For more information about implementing a **DriverEntry** routine, see Writing a DriverEntry Routine.

While it is possible to name this routine something other than **DriverEntry**, doing so is not recommended. The DDK-supplied build tools automatically inform the linker that the driver's entry point is called **DriverEntry**, so giving the routine another name requires you to modify the build tools. For more information about build tools, see Building a Driver.

Examples

To define a **DriverEntry** callback routine, you must first provide a function declaration that identifies the type of

callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

To define a **DriverEntry** callback routine, use the DRIVER_INITIALIZE type as shown in this code example:

```
DRIVER_INITIALIZE DriverEntry;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  DriverEntry(
    struct _DRIVER_OBJECT  *DriverObject,
    PUNICODE_STRING  RegistryPath
    )
  {
      // Function body
  }
```

The DRIVER_INITIALIZE function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the DRIVER_INITIALIZE function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Target Platform	Desktop
Header	wudfwdm.h (include Mcd.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Called at PASSIVE_LEVEL.

See Also

DRIVER_OBJECT

InitializeListHead function

1/25/2019 • 2 minutes to read • Edit Online

The **InitializeListHead** routine initializes a LIST_ENTRY structure that represents the head of a doubly linked list.

Syntax

```
void InitializeListHead(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to a LIST_ENTRY structure that serves as the list header.

Return Value

None

Remarks

The **InitializeListHead** routine sets the **Flink** and **Blink** members of *ListHead* to point to *ListHead*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InitializeListHead** can be running at any IRQL. If **InitializeListHead** is called at IRQL >= DISPATCH_LEVEL the storage for *ListHead* must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedInsertHeadList

ExInterlockedInsertTailList

ExInterlockedPopEntryList

ExInterlockedPushEntryList

ExInterlockedRemoveHeadList

InsertHeadList

InsertTailList

IsListEmpty

KeInitializeSpinLock

PopEntryList

PushEntryList

RemoveEntryList

RemoveHeadList

RemoveTailList

InitializeObjectAttributes macro

1/25/2019 • 2 minutes to read • Edit Online

The **InitializeObjectAttributes** macro initializes the opaque OBJECT_ATTRIBUTES structure, which specifies the properties of an object handle to routines that open handles.

Syntax

```
void InitializeObjectAttributes(
   p,
   n,
   a,
   r,
   s
);
```

Parameters

`p`

A pointer to the OBJECT_ATTRIBUTES structure to initialize.

`n`

A pointer to a Unicode string that contains the name of the object for which a handle is to be opened. This must either be a fully qualified object name, or a relative path name to the object directory specified by the RootDirectory parameter.

`a`

Specifies one or more of the following flags:

FLAG	DESCRIPTION
OBJ_INHERIT	This handle can be inherited by child processes of the current process.
OBJ_PERMANENT	This flag only applies to objects that are named within the object manager. By default, such objects are deleted when all open handles to them are closed. If this flag is specified, the object is not deleted when all open handles are closed. Drivers can use ZwMakeTemporaryObject to delete permanent objects.
OBJ_EXCLUSIVE	Only a single handle can be open for this object.
OBJ_CASE_INSENSITIVE	If this flag is specified, a case-insensitive comparison is used when matching the ObjectName parameter against the names of existing objects. Otherwise, object names are compared using the default system settings.

FLAG	DESCRIPTION
OBJ_OPENIF	If this flag is specified to a routine that creates objects, and that object already exists then the routine should open that object. Otherwise, the routine creating the object returns an NTSTATUS code of STATUS_OBJECT_NAME_COLLISION.
OBJ_KERNEL_HANDLE	Specifies that the handle can only be accessed in kernel mode.
OBJ_FORCE_ACCESS_CHECK	The routine opening the handle should enforce all access checks for the object, even if the handle is being opened in kernel mode.

`r`

A handle to the root object directory for the path name specified in the ObjectName parameter. If ObjectName is a fully qualified object name, RootDirectory is NULL. Use **ZwCreateDirectoryObject** to obtain a handle to an object directory.

`s`

Specifies a security descriptor to apply to an object when it is created. This parameter is optional. Drivers can specify NULL to accept the default security for the object. For more information, see the following Remarks section.

Return Value

None

Remarks

InitializeObjectAttributes initializes an OBJECT_ATTRIBUTES structure that specifies the properties of an object handle to be opened. The caller can then pass a pointer to this structure to a routine that actually opens the handle.

Driver routines that run in a process context other than that of the system process must set the OBJ_KERNEL_HANDLE flag for the *Attributes* parameter. This flag restricts the use of a handle opened for that object to processes running only in kernel mode. Otherwise, the handle can be accessed by the process in whose context the driver is running.

Note that **InitializeObjectAttributes** always sets the **SecurityQualityOfService** member of OBJECT_ATTRIBUTES to **NULL**. Drivers that require a non-**NULL** value can set **SecurityQualityOfService** directly.

Requirements

Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntdef.h)

See Also

ExCreateCallback

IoCreateFile

OBJECT_ATTRIBUTES

PsCreateSystemThread

SECURITY_DESCRIPTOR

UNICODE_STRING

ZwCreateDirectoryObject

ZwCreateFile

ZwCreateKey

ZwMakeTemporaryObject

ZwOpenFile

ZwOpenKey

ZwOpenSection

ZwOpenSymbolicLinkObject

InsertHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **InsertHeadList** routine inserts an entry at the head of a doubly linked list of LIST_ENTRY structures.

Syntax

```
void InsertHeadList(
  PLIST_ENTRY                 ListHead,
  __drv_aliasesMem PLIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that represents the head of the list.

`Entry`

Pointer to a LIST_ENTRY structure that represents the entry to be inserted into the list.

Return Value

None

Remarks

InsertHeadList updates *ListHead*->**Flink** to point to *Entry*. It updates *Entry*->**Flink** to point to the old first entry in the list, and sets *Entry*->**Blink** to *ListHead*. The **Blink** field of the original first entry is also updated to point to *Entry*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertHeadList** can be running at any IRQL. If **InsertHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

DDI compliance rules	IoAllocateFree, IoReuseIrp, MarkingQueuedIrps, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite

See Also

ExInterlockedInsertHeadList

InitializeListHead

InsertTailList

IsListEmpty

RemoveHeadList

RemoveTailList

InsertTailList function

1/25/2019 • 2 minutes to read • Edit Online

The **InsertTailList** routine inserts an entry at the tail of a doubly linked list of LIST_ENTRY structures.

Syntax

```
void InsertTailList(
  PLIST_ENTRY                 ListHead,
  __drv_aliasesMem PLIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that represents the head of the list.

`Entry`

Pointer to a LIST_ENTRY structure that represents the entry to be inserted in the list.

Return Value

None

Remarks

InsertTailList updates *ListHead*->**Blink** to point to *Entry*. It updates *Entry*->**Blink** to point to the old last entry in the list, and sets *Entry*->**Flink** to *ListHead*. The **Flink** of the previous last entry is updated to point to *Entry* as well.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertTailList** can be running at any IRQL. If **InsertTailList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedInsertTailList

InitializeListHead

InsertHeadList

IsListEmpty

RemoveHeadList

RemoveTailList

IsListEmpty function

1/25/2019 • 2 minutes to read • Edit Online

The **IsListEmpty** routine indicates whether a doubly linked list of LIST_ENTRY structures is empty.

Syntax

```
BOOLEAN IsListEmpty(
  const LIST_ENTRY *ListHead
);
```

Parameters

`ListHead`

Pointer to a LIST_ENTRY structure that represents the head of the list.

Return Value

IsListEmpty returns **TRUE** if there are currently no entries in the list and **FALSE** otherwise.

Remarks

IsListEmpty returns **TRUE** if *ListHead*->**Flink** refers back to *ListHead*.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **IsListEmpty** can be running at any IRQL. If **IsListEmpty** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

InitializeListHead

RemoveEntryList

RemoveHeadList

RemoveTailList

PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentActiveConditionCallback* callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.

Syntax

```
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK PoFxComponentActiveConditionCallback;

void PoFxComponentActiveConditionCallback(
  PVOID Context,
  ULONG Component
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

Specifies the index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Return Value

None.

Remarks

A component's hardware registers can be accessed only when the component is in the active condition. Do not rely on the Fx power state of a component to determine whether the component can be accessed. If the component is in the F0 state and is in the idle condition, the component might be about to switch to another Fx state.

To gain access to a component that is in the idle condition, the driver can call the PoFxActivateComponent routine to switch the component to the active condition. If the component is not already in the active condition when **PoFxActivateComponent** is called, PoFx initiates the transition to the active condition, and then calls the *ComponentActiveConditionCallback* routine to notify the driver when the transition is complete.

The driver receives a *ComponentActiveConditionCallback* callback whenever the component completes a transition to the active condition, regardless of whether the driver called **PoFxActivateComponent** to initiate the transition. For example, the Windows kernel might activate the component on behalf of the driver before sending a Plug and Play IRP or a system set-power IRP to the driver.

Examples

To define a *ComponentActiveConditionCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentActiveConditionCallback* callback routine that is named `MyComponentActiveConditionCallback`, use the PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK MyComponentActiveConditionCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentActiveConditionCallback(
    PVOID Context,
    ULONG Component
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_ACTIVE_CONDITION_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxActivateComponent

PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentCriticalTransitionCallback* callback routine handles a transition of the specified component between the F0 (fully on) and low-power F*x* component power states.

Syntax

```
PO_FX_COMPONENT_CRITICAL_TRANSITION_CALLBACK PoFxComponentCriticalTransitionCallback;

void PoFxComponentCriticalTransitionCallback(
  PVOID Context,
  ULONG Component,
  BOOLEAN Active
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. This context is driver-defined and is opaque to PoFx. The driver specified this pointer in the **DeviceContext** member of the PO_FX_CORE_DEVICE structure that the driver used to register the device with the Windows power management framework (PoFx).

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_CORE_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

`Active`

Indicates whether this notification is for a transition to the F0 component power state. If TRUE, the component has just completed a transition from a low-power F*x* state to F0. If FALSE, the component is about to start a transition from F0 to a low-power F*x* state.

Return Value

None.

Remarks

This callback routine is implemented by a device driver, and is called by PoFx. PoFx calls this routine to notify the driver of power transitions of device components.

The **ComponentCriticalTransitionCallback** member of the PO_FX_CORE_DEVICE structure is a pointer to a *ComponentCriticalTransitionCallback* callback routine. A pointer to a **PO_FX_CORE_DEVICE** structure is passed as an input parameter in the PoFxRegisterCoreDevice call that registers a core device with the Windows power management framework (PoFx).

To handle calls to this routine, the device driver saves or restores the register state or other hardware context of the

specified component in the device.

If *Active* = TRUE, PoFx has called this routine *after* the component completed a transition from a low-power F*x* state to F0. In this call, the routine restores the previously saved hardware context of the component.

If *Active* = FALSE, PoFx has called this routine *before* the component starts a pending transition from F0 to a low-power F*x* state. In this call, the routine saves the component's hardware context so that this context can later be restored.

For more information about F*x* component power states, see Component-Level Power Management.

A PEP can call this routine at IRQL TBD.

Requirements

Minimum supported client	Supported starting with Windows 10.
Target Platform	Windows
Header	wudfwdm.h (include Wudfwdm.h)

See Also

PO_FX_CORE_DEVICE

PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *ComponentIdleConditionCallback* callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.

Syntax

```
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK PoFxComponentIdleConditionCallback;

void PoFxComponentIdleConditionCallback(
  PVOID Context,
  ULONG Component
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

The index that identifies the component. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N–1.

Return Value

None.

Remarks

When the driver no longer needs to access a component that is in the active condition, the driver should call the PoFxIdleComponent routine to switch the component to the idle condition. In response to the **PoFxIdleComponent** call, PoFx initiates the transition to the idle condition, and then calls the *ComponentIdleConditionCallback* routine to notify the driver when this transition is complete.

The driver must call **PoFxCompleteIdleCondition** in response to every *ComponentIdleConditionCallback* callback. The **PoFxCompleteIdleCondition** call can occur either during the *ComponentIdleConditionCallback* callback or after the callback returns. The driver should complete any work that requires access to the hardware component before the call to **PoFxCompleteIdleCondition**. After this call, the component might not stay in the F0 power state.

A component can be safely accessed only when the component is in the active condition. Do not rely on the Fx power state of a component to determine whether the component can be accessed. If the component is in the F0

state and is in the idle condition, the component might be about to switch to another Fx state.

Examples

To define a *ComponentIdleConditionCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentIdleConditionCallback* callback routine that is named `MyComponentIdleConditionCallback`, use the PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK MyComponentIdleConditionCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentIdleConditionCallback(
    PVOID Context,
    ULONG Component
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_IDLE_CONDITION_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxCompleteIdleCondition

PO_FX_COMPONENT_IDLE_STATE_CALLBACK callback function

1/25/2019 • 4 minutes to read • Edit Online

The *ComponentIdleStateCallback* callback routine notifies the driver of a pending change to the Fx power state of the specified component.

Syntax

```
PO_FX_COMPONENT_IDLE_STATE_CALLBACK PoFxComponentIdleStateCallback;

void PoFxComponentIdleStateCallback(
  PVOID Context,
  ULONG Component,
  ULONG State
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

`Component`

Specifies the component number. This parameter is an index into the **Components** array in the **PO_FX_DEVICE** structure that the device driver used to register the device with PoFx. If the **Components** array contains N elements, component indexes range from 0 to N-1.

`State`

Specifies the new Fx power state that the component will change to. If this parameter is zero, the new state is F0; if this parameter is one, the new state is F1; and so on.

Return Value

None.

Remarks

When PoFx calls the driver's *ComponentIdleStateCallback* routine, the driver might need to prepare for the pending Fx state change. After any necessary preparations are completed, the driver must call the PoFxCompleteIdleState routine to notify PoFx that the driver completed its response to the *ComponentIdleStateCallback* callback. The **PoFxCompleteIdleState** call can occur either before or after the *ComponentIdleStateCallback* routine returns.

If the component is to switch from F0 to a low-power Fx state in which the device will lose the hardware state of

the component, the driver must save the component's hardware state before the transition to the new Fx state occurs. If the component is to switch from a low-power Fx state to F0, and the hardware state was previously saved, the driver should restore the hardware state after power is restored to the component.

For some devices, the power state of a component might be controlled in the device hardware. For these devices, the driver is typically responsible for changing the power state of the component. For other devices, the power state of a component might be controlled by hardware that is external to the device, and the driver typically must rely on PoFx to configure the power state of the component.

If the driver is responsible for configuring the power state of the component, the driver should change the power state in response to the *ComponentIdleStateCallback* callback. For a transition from F0 to a low-power Fx state, the component is in F0 on entry to the *ComponentIdleStateCallback* routine, and the driver must switch the component the new Fx state before calling **PoFxCompleteIdleState**. For a transition from a low-power Fx state to F0, the component is in the low-power Fx state on entry to the *ComponentIdleStateCallback* routine, and the driver must switch the component to F0 before calling **PoFxCompleteIdleState**.

If the driver is not responsible for configuring the power state of the component, the driver should assume that the component is always in the F0 state on entry to the *ComponentIdleStateCallback* routine. For a pending transition from F0 to a low-power Fx state, the transition to the new Fx state does not occur until after the driver calls **PoFxCompleteIdleState**. For a transition from a low-power Fx state to F0, the transition to F0 occurs before the *ComponentIdleStateCallback* routine is called.

PoFx never calls the *ComponentIdleStateCallback* routine to switch directly from one low-power Fx state to another low-power Fx state. For example, PoFx might need to switch a component from one low-power Fx state to another in response to a call to the PoFxSetComponentLatency, PoFxSetComponentResidency, or PoFxSetComponentWake routine. In this case, PoFx first calls the *ComponentIdleStateCallback* routine to switch from the old Fx state to F0, and then calls the *ComponentIdleStateCallback* routine a second time to switch from F0 to the new Fx state.

Examples

To define a *ComponentIdleStateCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *ComponentIdleStateCallback* callback routine that is named `MyComponentIdleStateCallback`, use the PO_FX_COMPONENT_IDLE_STATE_CALLBACK type as shown in this code example:

```
PO_FX_COMPONENT_IDLE_STATE_CALLBACK MyComponentIdleStateCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyComponentIdleStateCallback(
    PVOID Context,
    ULONG Component,
    ULONG State
    )
  {
      // Function body
  }
```

The PO_FX_COMPONENT_IDLE_STATE_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_COMPONENT_IDLE_STATE_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported in Windows 8 and later versions of Windows.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxCompleteIdleState

PoFxSetComponentLatency

PoFxSetComponentResidency

PoFxSetComponentWake

PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DevicePowerNotRequiredCallback* callback routine notifies the device driver that the device is not required to stay in the D0 power state.

Syntax

```
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK PoFxDevicePowerNotRequiredCallback;

void PoFxDevicePowerNotRequiredCallback(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

Return Value

None.

Remarks

When PoFx calls the driver's *DevicePowerNotRequiredCallback* routine, the driver must first decide whether to initiate a transition to a low-power Dx state (by sending an IRP_MN_SET_POWER request down the device stack) or to remain in the D0 state. Next, without waiting for any Dx transition to complete, the driver must call the PoFxCompleteDevicePowerNotRequired routine to notify PoFx that the driver completed its response to the *DevicePowerNotRequiredCallback* callback. The driver can call **PoFxCompleteDevicePowerNotRequired** before or after the *DevicePowerNotRequiredCallback* routine returns.

When a device is in the D0 state, and the Fx state or active/idle condition of a component in the device changes, PoFx evaluates whether the device can enter a low-power Dx state or must remain in the D0 state. If the device can enter a low-power Dx state, PoFx calls the driver's *DevicePowerNotRequiredCallback* routine.

If the device enters a low-power Dx state in response to a *DevicePowerNotRequiredCallback* callback, but PoFx later determines that the device must enter the D0 state, PoFx calls the driver's DevicePowerRequiredCallback routine. In response to this call, the device must enter the D0 state.

Examples

To define a *DevicePowerNotRequiredCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV),

and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DevicePowerNotRequiredCallback* callback routine that is named `MyDevicePowerNotRequiredCallback`, use the PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK type as shown in this code example:

```
PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK MyDevicePowerNotRequiredCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDevicePowerNotRequiredCallback(
    PVOID Context
    )
  {
      // Function body
  }
```

The PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_DEVICE_POWER_NOT_REQUIRED_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

DevicePowerRequiredCallback

IRP_MN_SET_POWER

PO_FX_DEVICE

PoFxCompleteDevicePowerNotRequired

PoFxRegisterDevice

PO_FX_DEVICE_POWER_REQUIRED_CALLBACK callback function

1/25/2019 • 2 minutes to read • Edit Online

The *DevicePowerRequiredCallback* callback routine notifies the device driver that the device must enter and remain in the D0 power state.

Syntax

```
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK PoFxDevicePowerRequiredCallback;

void PoFxDevicePowerRequiredCallback(
  PVOID Context
)
{...}
```

Parameters

`Context`

A pointer to the device context. The device driver uses this context to store information about the current power state of the device. The device driver specified this pointer in the **DeviceContext** member of the PO_FX_DEVICE structure that the driver used to register the device with the power management framework (PoFx). This context is opaque to PoFx.

Return Value

None.

Remarks

After PoFx calls the driver's *DevicePowerRequiredCallback* routine, the driver must initiate a transition to the D0 state by sending a D0 IRP (an IRP_MN_SET_POWER request) down the device stack, if the device is not already in the D0 state. After the driver completes (and all lower drivers complete) handling of this request, the driver must call the PoFxReportDevicePoweredOn routine. The **PoFxReportDevicePoweredOn** call notifies PoFx that the device completed its response to the *DevicePowerRequiredCallback* call. The driver can call **PoFxReportDevicePoweredOn** during or after the *DevicePowerRequiredCallback* call.

Each time the *DevicePowerRequiredCallback* routine is called, the driver must respond by calling **PoFxReportDevicePoweredOn**. The **PoFxReportDevicePoweredOn** call can occur either during or after the *DevicePowerRequiredCallback* call. The driver must call **PoFxReportDevicePoweredOn** regardless of whether no D0 IRP is required, the D0 IRP request fails, or the dispatched D0 IRP succeeds or fails.

PoFx calls the *DevicePowerRequiredCallback* routine only if PoFx previously called the driver's DevicePowerNotRequiredCallback routine to notify the driver that the device was not required to stay in the D0 state.

The call to the *DevicePowerRequiredCallback* routine can occur at any time after the driver calls PoFxCompleteDevicePowerNotRequired to complete a *DevicePowerNotRequiredCallback* callback. The *DevicePowerRequiredCallback* call does not necessarily occur as a result of an action taken by the driver, such as

requesting that a component change to the active condition.

Examples

To define a *DevicePowerRequiredCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *DevicePowerRequiredCallback* callback routine that is named `MyDevicePowerRequiredCallback`, use the PO_FX_DEVICE_POWER_REQUIRED_CALLBACK type as shown in this code example:

```
PO_FX_DEVICE_POWER_REQUIRED_CALLBACK MyDevicePowerRequiredCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
VOID
  MyDevicePowerRequiredCallback(
    PVOID Context
    )
  {
      // Function body
  }
```

The PO_FX_DEVICE_POWER_REQUIRED_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_DEVICE_POWER_REQUIRED_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

DevicePowerNotRequiredCallback

PO_FX_DEVICE

PoFxCompleteDevicePowerNotRequired

PoFxRegisterDevice

PoFxReportDevicePoweredOn

PO_FX_POWER_CONTROL_CALLBACK callback function

1/25/2019 • 3 minutes to read • Edit Online

The *PowerControlCallback* callback routine performs a power control operation that is requested by the power management framework (PoFx).

Syntax

```
PO_FX_POWER_CONTROL_CALLBACK PoFxPowerControlCallback;

NTSTATUS PoFxPowerControlCallback(
  PVOID DeviceContext,
  LPCGUID PowerControlCode,
  PVOID InBuffer,
  SIZE_T InBufferSize,
  PVOID OutBuffer,
  SIZE_T OutBufferSize,
  PSIZE_T BytesReturned
)
{...}
```

Parameters

`DeviceContext`

`PowerControlCode`

A pointer to the power control code. This code is a GUID value that specifies the requested operation.

`InBuffer`

A pointer to the buffer that contains the input data for the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *InBuffer* parameter is optional and can be specified as NULL if the specified operation requires no input data.

`InBufferSize`

The size, in bytes, of the input buffer that is pointed to by the *InBuffer* parameter. If *InBuffer* is NULL, this parameter is zero.

`OutBuffer`

A pointer to the buffer to which the callback routine writes the output data from the operation. The format for the data in this buffer depends on the power control code specified by the *PowerControlCode* parameter. The *OutBuffer* parameter is optional and can be specified as NULL if the specified operation produces no output data.

`OutBufferSize`

The size, in bytes, of the output buffer that is pointed to by the *OutBuffer* parameter. If *OutBuffer* is NULL, this parameter is zero.

`BytesReturned`

A pointer to a location into which the routine writes the number of bytes of data that were written to the buffer that is pointed to by *OutBuffer*. The number of bytes written must be less than or equal to *OutBufferSize*. This parameter is optional and can be specified as NULL if the caller does not need to know how many bytes were written to the output buffer.

Return Value

The *PowerControlCallback* routine returns STATUS_SUCCESS if the call is successful. Otherwise, it returns an appropriate error code.

Remarks

PoFx calls this routine to send a power control request directly to the device driver. A power control request is similar to an I/O control request (IOCTL). Unlike an IOCTL, however, a power control request is sent directly to the driver and is not observed by other device drivers in the device stack. During a *PowerControlCallback* call, the driver synchronously performs the requested operation.

This routine is optional. A device driver that does not support power control operations is not required to implement a *PowerControlCallback* routine.

The device driver can call the PoFxPowerControl routine to send a power control request to PoFx.

For more information about power control requests, see PoFxPowerControl.

Examples

To define a *PowerControlCallback* callback routine, you must first provide a function declaration that identifies the type of callback routine you're defining. Windows provides a set of callback function types for drivers. Declaring a function using the callback function types helps Code Analysis for Drivers, Static Driver Verifier (SDV), and other verification tools find errors, and it's a requirement for writing drivers for the Windows operating system.

For example, to define a *PowerControlCallback* callback routine that is named `MyPowerControlCallback`, use the PO_FX_POWER_CONTROL_CALLBACK type as shown in this code example:

```
PO_FX_POWER_CONTROL_CALLBACK MyPowerControlCallback;
```

Then, implement your callback routine as follows:

```
_Use_decl_annotations_
NTSTATUS
  MyPowerControlCallback(
    PVOID Context,
    LPCGUID PowerControlCode,
    PVOID InBuffer,
    SIZE_T InBufferSize,
    PVOID OutBuffer,
    SIZE_T OutBufferSize,
    PSIZE_T BytesReturned
    )
  {
      // Function body
  }
```

The PO_FX_POWER_CONTROL_CALLBACK function type is defined in the Wdm.h header file. To more accurately identify errors when you run the code analysis tools, be sure to add the *Use_decl_annotations* annotation to your function definition. The *Use_decl_annotations* annotation ensures that the annotations that are applied to the PO_FX_POWER_CONTROL_CALLBACK function type in the header file are used. For more information about the requirements for function declarations, see Declaring Functions by Using Function Role Types for WDM Drivers. For information about *Use_decl_annotations*, see Annotating Function Behavior.

Requirements

Minimum supported client	Supported starting with Windows 8.
Target Platform	Desktop
Header	wudfwdm.h (include Wudfwdm.h)
IRQL	Called at IRQL <= DISPATCH_LEVEL.

See Also

PO_FX_DEVICE

PoFxPowerControl

PopEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **PopEntryList** routine removes the first entry from a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
PSINGLE_LIST_ENTRY PopEntryList(
  PSINGLE_LIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the SINGLE_LIST_ENTRY structure that represents the head of the list. On return, *ListHead*->**Next** points to the beginning of the list with the first entry removed.

Return Value

PopEntryList returns a pointer to the entry removed from the list, or **NULL** if the list is currently empty.

Remarks

PopEntryList removes the first entry from the list by setting *ListHead*->**Next** to point to the second entry in the list.

For information about using this routine when implementing a singly linked list, see Singly and Doubly Linked Lists.

Callers of **PopEntryList** can be running at any IRQL. If **PopEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (see Remarks section)

See Also

ExInterlockedPopEntryList

PushEntryList

PushEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **PushEntryList** routine inserts an entry at the beginning of a singly linked list of SINGLE_LIST_ENTRY structures.

Syntax

```
void PushEntryList(
  PSINGLE_LIST_ENTRY                  ListHead,
  __drv_aliasesMem PSINGLE_LIST_ENTRY Entry
);
```

Parameters

`ListHead`

Pointer to the SINGLE_LIST_ENTRY structure that serves as the list header.

`Entry`

Pointer to SINGLE_LIST_ENTRY structure that represents the entry to be inserted on the list.

Return Value

None

Remarks

PushEntryList sets *ListHead*->**Next** to *Entry*, and *Entry*->**Next** to point to the old first entry of the list.

For information about using this routine when implementing a singly linked list, see Singly and Doubly Linked Lists.

Callers of **PushEntryList** can be running at any IRQL. If **PushEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

ExInterlockedPushEntryList

PopEntryList

READ_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_UCHAR** routine reads a number of bytes from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of UCHAR values is read.

`Count`

Specifies the number of bytes to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **READ_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_ULONG** routine reads a number of ULONG values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of ULONG values is read.

`Count`

Specifies the number of ULONG values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **READ_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_BUFFER_USHORT** routine reads a number of USHORT values from the specified register address into a buffer.

Syntax

```
void READ_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer into which an array of USHORT values is read.

`Count`

Specifies the number of USHORT values to be read into the buffer.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **READ_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_UCHAR** routine reads a byte from the specified register address.

Syntax

```
UCHAR READ_REGISTER_UCHAR(
  volatile UCHAR *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_UCHAR returns the byte read from the specified register address.

Remarks

Callers of **READ_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_ULONG** routine reads a ULONG value from the specified register address.

Syntax

```
ULONG READ_REGISTER_ULONG(
  volatile ULONG *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_ULONG returns the ULONG value read from the specified register address.

Remarks

Callers of **READ_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

READ_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **READ_REGISTER_USHORT** routine reads a USHORT value from the specified register address.

Syntax

```
USHORT READ_REGISTER_USHORT(
  volatile USHORT *Register
);
```

Parameters

`Register`

Return Value

READ_REGISTER_USHORT returns the USHORT value read from the specified register address.

Remarks

Callers of **READ_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

RemoveEntryList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveEntryList** routine removes an entry from a doubly linked list of LIST_ENTRY structures.

Syntax

```
BOOLEAN RemoveEntryList(
  PLIST_ENTRY Entry
);
```

Parameters

`Entry`

Pointer to the LIST_ENTRY structure that represents the entry to be removed.

Return Value

RemoveEntryList returns **TRUE** if, after removal of the designated entry, the list is empty. Otherwise, the routine returns **FALSE** to indicate that the resulting list still contains one or more entries. For information, see Remarks.

> **Note** Before Microsoft Windows Server 2003, **RemoveEntryList** was defined to return VOID, not BOOLEAN.

Remarks

RemoveEntryList removes the entry by setting the **Flink** member of the entry before *Entry* to point to the entry after *Entry*, and the **Blink** member of the entry after *Entry* to point to the entry before *Entry*.

The return value can be used to detect when the last entry is removed from the list. An empty list consists of a list head only and no list entries.

> **Note** Typically, *Entry* points to an entry in a list and not to the list head. However, *Entry* can point to a list head, in which case the routine removes the list head from the list to produce a headless list. When **RemoveEntryList** is used in this way, the return value should typically be ignored. To determine whether a list is empty, use the IsListEmpty routine.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

In Windows XP and Windows 2000, **RemoveEntryList** did not return a value. Starting with Windows Server 2003, **RemoveEntryList** returns a BOOLEAN value.

Callers of **RemoveEntryList** can be running at any IRQL. If **RemoveEntryList** is called at IRQL >= DISPATCH_LEVEL, the storage for the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)
DDI compliance rules	DoubleExFreePool

See Also

InitializeListHead

IsListEmpty

RemoveHeadList

RemoveTailList

RemoveHeadList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveHeadList** routine removes an entry from the beginning of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY RemoveHeadList(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that serves as the list header.

Return Value

RemoveHeadList returns a pointer to the entry removed from the list. If the list is empty, **RemoveHeadList** returns *ListHead*.

Remarks

RemoveHeadList removes the first entry from the list by setting *ListHead*->**Flink** to point to the second entry in the list. The routine sets the **Blink** member of the second entry to *ListHead*. In the event the list is empty, this is effectively a no-op.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **RemoveHeadList** can be running at any IRQL. If **RemoveHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

DDI compliance rules	CancelSpinLock, CompleteRequest, DoubleCompletion, IoAllocateFree, IoReuseIrp, IrpProcessingComplete, MarkingInterlockedQueuedIrps, MarkingQueuedIrps, MarkIrpPending, MarkIrpPending2, PendedCompletedRequest, PendedCompletedRequest2, PendedCompletedRequest3, PendedCompletedRequestEx, RemoveLock, RemoveLockCheck, RemoveLockForward, RemoveLockForward2, RemoveLockForwardDeviceControl, RemoveLockForwardDeviceControl2, RemoveLockForwardDeviceControlInternal, RemoveLockForwardDeviceControlInternal2, RemoveLockForwardRead, RemoveLockForwardRead2, RemoveLockForwardWrite, RemoveLockForwardWrite2, RemoveLockMnRemove, RemoveLockMnSurpriseRemove, RemoveLockRelease2, RemoveLockReleaseCleanup, RemoveLockReleaseClose, RemoveLockReleaseCreate, RemoveLockReleaseDeviceControl, RemoveLockReleaseInternalDeviceControl, RemoveLockReleasePnp, RemoveLockReleasePower, RemoveLockReleaseRead, RemoveLockReleaseShutdown, RemoveLockReleaseSystemControl, RemoveLockReleaseWrite, InvalidReqAccessLocal, DoubleExFreePool, Init_NdisAllocateIoWorkItem

See Also

ExInterlockedRemoveHeadList

InitializeListHead

IsListEmpty

RemoveEntryList

RemoveTailList

RemoveTailList function

1/25/2019 • 2 minutes to read • Edit Online

The **RemoveTailList** routine removes an entry from the end of a doubly linked list of LIST_ENTRY structures.

Syntax

```
PLIST_ENTRY RemoveTailList(
  PLIST_ENTRY ListHead
);
```

Parameters

`ListHead`

Pointer to the LIST_ENTRY structure that serves as the list header.

Return Value

RemoveTailList returns a pointer to the entry that was at the tail of the list. If the list is empty, **RemoveTailList** returns *ListHead*.

Remarks

RemoveTailList removes the last entry from the list by setting *ListHead*->**Blink** to point to the second-to-last entry in the list. The routine sets the **Flink** member of the new first entry to *ListHead*. In the event the list is empty, this is effectively a no-op.

For information about using this routine when implementing a doubly linked list, see Singly and Doubly Linked Lists.

Callers of **InsertHeadList** can be running at any IRQL. If **InsertHeadList** is called at IRQL >= DISPATCH_LEVEL, the storage for *ListHead* and the list entries must be resident.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
IRQL	Any level (See Remarks section)

See Also

InitializeListHead

IsListEmpty

RemoveEntryList

RemoveHeadList

RtlAppendUnicodeStringToString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAppendUnicodeStringToString** routine concatenates two Unicode strings.

Syntax

```
NTSYSAPI NTSTATUS RtlAppendUnicodeStringToString(
  PUNICODE_STRING  Destination,
  PCUNICODE_STRING Source
);
```

Parameters

`Destination`

Pointer to a buffered Unicode string.

`Source`

Pointer to the buffered string to be concatenated.

Return Value

RtlAppendUnicodeStringToString can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The source string was successfully appended to the destination counted string. The destination string length is updated to include the appended bytes.
STATUS_BUFFER_TOO_SMALL	The destination string length is too small to allow the source string to be concatenated. Accordingly, the destination string length is not updated.

Remarks

RtlAppendUnicodeStringToString copies bytes from the source up to the length of the destination buffer.

The *Destination* and *Source* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	See Remarks section.

See Also

RtlAppendUnicodeToString

RtlAppendUnicodeToString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlAppendUnicodeToString** routine concatenates the supplied Unicode string to a buffered Unicode string.

Syntax

```
NTSYSAPI NTSTATUS RtlAppendUnicodeToString(
  PUNICODE_STRING Destination,
  PCWSTR          Source
);
```

Parameters

`Destination`

A pointer to the buffered string.

`Source`

A pointer to the null-terminated string to append to the *Destination* string.

Return Value

RtlAppendUnicodeToString can return one of the following:

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The source string was successfully appended to the destination string, and the destination string length was updated to include the appended bytes.
STATUS_BUFFER_TOO_SMALL	The destination string buffer is too small to allow the source string to be appended, so the destination string length is not updated and the contents of the destination string buffer are not changed.

Remarks

RtlAppendUnicodeToString appends the source string to the end of the destination string.

If the destination buffer is not large enough to contain the string that would result from concatenating the source and destination strings (not including the null character that terminates the source string), the routine returns STATUS_BUFFER_TOO_SMALL and leaves the destination string unchanged.

The *Destination* buffer must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	See Remarks section.

See Also

RtlAppendUnicodeStringToString

RtlCompareUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCompareUnicodeString** routine compares two Unicode strings.

Syntax

```
NTSYSAPI LONG RtlCompareUnicodeString(
  PCUNICODE_STRING String1,
  PCUNICODE_STRING String2,
  BOOLEAN          CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first string.

`String2`

Pointer to the second string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlCompareUnicodeString returns a signed value that gives the results of the comparison:

RETURN CODE	DESCRIPTION
Zero	*String1* equals *String2*.
< Zero	*String1* is less than *String2*.
> Zero	*String1* is greater than *String2*.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlCompareString

RtlEqualString

RtlCopyUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlCopyUnicodeString** routine copies a source string to a destination string.

Syntax

```
NTSYSAPI VOID RtlCopyUnicodeString(
  PUNICODE_STRING  DestinationString,
  PCUNICODE_STRING SourceString
);
```

Parameters

`DestinationString`

A pointer to the destination string buffer. This parameter points to a UNICODE_STRING structure.

`SourceString`

A pointer to the source string buffer. This parameter points to a **UNICODE_STRING** structure.

Return Value

None

Remarks

If *SourceString* is **NULL**, this routine sets the **Length** field of the structure pointed to by *DestinationString* to zero.

This routine does not modify the **MaximumLength** and **Buffer** fields of the structure pointed to by *DestinationString*.

The number of bytes copied from the source string is either the source string length (specified by the **Length** member of the structure pointed to by *SourceString*) or the maximum length of the destination string (specified by the **MaximumLength** member of the structure pointed to by *DestinationString*), whichever is smaller.

The caller must properly initialize all members of the structure pointed to by *DestinationString* before calling **RtlCopyUnicodeString**. Failure to initialize the **Length** or the **MaximumLength** member before calling this routine can cause a buffer overrun.

The *DestinationString* and *SourceString* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal

Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	Any level (See Remarks section)

See Also

RtlCopyString

UNICODE_STRING

RtlEqualUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlEqualUnicodeString** routine compares two Unicode strings to determine whether they are equal.

Syntax

```
NTSYSAPI BOOLEAN RtlEqualUnicodeString(
  PCUNICODE_STRING String1,
  PCUNICODE_STRING String2,
  BOOLEAN          CaseInSensitive
);
```

Parameters

`String1`

Pointer to the first Unicode string.

`String2`

Pointer to the second Unicode string.

`CaseInSensitive`

If **TRUE**, case should be ignored when doing the comparison.

Return Value

RtlEqualUnicodeString returns **TRUE** if the two Unicode strings are equal; otherwise, it returns **FALSE**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlEqualString

RtlFreeUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlFreeUnicodeString** routine releases storage that was allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Syntax

```
NTSYSAPI VOID RtlFreeUnicodeString(
  PUNICODE_STRING UnicodeString
);
```

Parameters

`UnicodeString`

Pointer to the Unicode string buffer previously allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Return Value

None

Remarks

This routine does not release the ANSI string buffer passed to **RtlAnsiStringToUnicodeString**.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	"<=APC_LEVEL"

See Also

RtlAnsiStringToUnicodeString

RtlUpcaseUnicodeString

RtlInitUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Syntax

```
NTSYSAPI VOID RtlInitUnicodeString(
  PUNICODE_STRING         DestinationString,
  __drv_aliasesMem PCWSTR SourceString
);
```

Parameters

`DestinationString`

For more information, see the WdmlibRtlInitUnicodeStringEx function.

`SourceString`

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Return Value

For more information, see the WdmlibRtlInitUnicodeStringEx function.

Remarks

The **RTL_CONSTANT_STRING** macro creates a string or Unicode string structure to hold a counted string.

```
STRING RTL_CONSTANT_STRING(
  [in]  PCSZ SourceString
);

UNICODE_STRING RTL_CONSTANT_STRING(
  [in]  PCWSTR SourceString
);
```

RTL_CONSTANT_STRING returns either a string structure or Unicode string structure.

The **RTL_CONSTANT_STRING** macro replaces the RtlInitAnsiString, RtlInitString, and **RtlInitUnicodeString** routines when passing a constant string.

You can use **RTL_CONSTANT_STRING** to initialize global variables.

Requirements

Minimum supported client	Available starting with Windows 2000.

Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	For more information, see the Remarks section of the WdmlibRtlInitUnicodeStringEx function.

See Also

RtlUnicodeStringInit

RtlUnicodeStringInitEx

UNICODE_STRING

WdmlibRtlInitUnicodeStringEx

RtlInt64ToUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlInt64ToUnicodeString** routine converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.

Syntax

```
NTSYSAPI NTSTATUS RtlInt64ToUnicodeString(
  ULONGLONG       Value,
  ULONG           Base,
  PUNICODE_STRING String
);
```

Parameters

`Value`

Specifies the ULONGLONG value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to the UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlInt64ToUnicodeString returns an NTSTATUS value. Possible return values include :

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The routine successfully converted *Value* to a Unicode string.

STA TUS _BU FFE R_O VER FLO W	*Value* is too large to convert, or the **UNICODE_STRING** structure is too small to hold the result.
STA TUS _IN VAL ID_ PAR AM ETE R	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlIntegerToUnicodeString function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIntegerToUnicodeString** routine converts an unsigned integer value to a null-terminated string of one or more Unicode characters in the specified base.

Syntax

```
NTSYSAPI NTSTATUS RtlIntegerToUnicodeString(
  ULONG           Value,
  ULONG           Base,
  PUNICODE_STRING String
);
```

Parameters

`Value`

Specifies the ULONG value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to a UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlIntegerToUnicodeString returns an NTSTATUS value. Possible return values include:

RETURN CODE	DESCRIPTION
STA TUS _SU CCE SS	The routine successfully converted *Value* to a Unicode string.

STA TUS _BU FFE R_O VER FLO W	*Value* is too large to convert, or the **UNICODE_STRING** structure is too small to hold the result.
STA TUS _IN VAL ID_ PAR AM ETE R	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlIntPtrToUnicodeString macro

1/25/2019 • 2 minutes to read • Edit Online

The **RtlIntPtrToUnicodeString** routine converts a specified ULONG_PTR value to a Unicode string that represents the value in a specified base.

Syntax

```
void RtlIntPtrToUnicodeString(
   Value,
   Base,
   String
);
```

Parameters

`Value`

Specifies the ULONG_PTR value to convert.

`Base`

Specifies the base to use when converting *Value* to a string. The possible values are:

VALUE	BASE
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

`String`

Pointer to a UNICODE_STRING structure that receives the string representation of *Value*. The buffer specified by the *Buffer* of *String* must be large enough to hold the result.

Return Value

RtlIntPtrToUnicodeString returns an NTSTATUS value. Possible return values include :

RETURN CODE	DESCRIPTION
STATUS_SUCCESS	The routine successfully converted Value to a Unicode string.
STATUS_BUFFER_OVERFLOW	Value is too large to convert, or the UNICODE_STRING structure is too small to hold the result.

RETURN CODE	DESCRIPTION
STATUS_INVALID_PARAMETER	The specified code base is not valid. The only valid values are 0, 2, 8, 10, and 16.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Desktop
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	PASSIVE_LEVEL

See Also

RtlAppendUnicodeStringToString

RtlIntegerToUnicodeString

RtlUnicodeStringToInteger

UNICODE_STRING

RtlUnicodeStringToInteger function

1/25/2019 • 2 minutes to read • Edit Online

The **RtlUnicodeStringToInteger** routine converts a Unicode string representation of a number to the equivalent integer value.

Syntax

```
NTSYSAPI NTSTATUS RtlUnicodeStringToInteger(
  PCUNICODE_STRING String,
  ULONG            Base,
  PULONG           Value
);
```

Parameters

`String`

A pointer to a UNICODE_STRING structure that contains the number representation to convert to the equivalent integer value.

`Base`

A numeric value that indicates the base (or radix) of the number that the Unicode string represents. This parameter value is optional and can be set to zero.

If *Base* is zero, **RtlUnicodeStringToInteger** checks the prefix of the Unicode string to determine the base of the number:

- If the prefix is "0x", **RtlUnicodeStringToInteger** interprets the number in the string as a hexadecimal integer.
- If the prefix is "0o", **RtlUnicodeStringToInteger** interprets the number in the string as an octal integer.
- If the prefix is "0b", **RtlUnicodeStringToInteger** interprets the number in the string as a binary integer.

If the Unicode string does not contain any of these prefixes, **RtlUnicodeStringToInteger** treats the string as a base-10 integer.

`Value`

A pointer to a ULONG variable to which **RtlUnicodeStringToInteger** writes the integer value that results from conversion of the Unicode string.

Return Value

If the conversion is successful, the **RtlUnicodeStringToInteger** routine returns STATUS_SUCCESS and sets **Value* to the integer value represented by the number in the Unicode string. If the string is not empty, but does not start with a valid number representation, the routine returns STATUS_SUCCESS and sets **Value* to zero. If the string is empty, the routine fails and returns STATUS_INVALID_PARAMETER.

Remarks

This routine skips any white space at the start of the input string to find the start of the number.

If the first non-white space character in the string is a hyphen (-), the integer value written to **Value* is negative; otherwise, if the first character is a "+" or there is no sign character, the integer value written to **Value* is positive.

If the string does not contain a valid number, or if the first digit in the string is preceded by a non-white space character other than '+' or '-', the routine sets the output value to zero and returns STATUS_SUCCESS.

A substring that contains one or more valid digits is terminated by any character that is not a valid digit. For example, if *Base* = 2, valid digits are '0' and '1'. If *Base* = 8, valid digits are '0' to '7'. If *Base* = 10, valid digits are '0' to '9'. If *Base* = 16, valid digits are '0' to '9', 'a' to 'f', and 'A' to 'F'.

The following table contains examples of output values that result from various combinations of input strings and *Base* parameter values.

INPUT STRING	BASE	OUTPUT VALUE
"123"	10	123
" -345"	10	-345
"xyz"	10	0
" +678abc"	10	678
" +678abc"	16	6785724
"007"	10	7
"789"	8	7
"FGH"	16	15
" "	10	0

A related routine, RtlIntegerToUnicodeString, converts an integer value to the equivalent Unicode string representation.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe (kernel mode); Ntdll.dll (user mode)
IRQL	PASSIVE_LEVEL

See Also

RtlIntegerToUnicodeString

UNICODE_STRING

WRITE_REGISTER_BUFFER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_UCHAR** routine writes a number of bytes from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_UCHAR(
  volatile UCHAR *Register,
  PUCHAR         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of UCHAR values is to be written.

`Count`

Specifies the number of bytes to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of **WRITE_REGISTER_BUFFER_UCHAR** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_ULONG** routine writes a number of ULONG values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_ULONG(
  volatile ULONG *Register,
  PULONG         Buffer,
  ULONG          Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of ULONG values is to be written.

`Count`

Specifies the number of ULONG values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of **WRITE_REGISTER_BUFFER_ULONG** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_BUFFER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_BUFFER_USHORT** routine writes a number of USHORT values from a buffer to the specified register.

Syntax

```
void WRITE_REGISTER_BUFFER_USHORT(
  volatile USHORT *Register,
  PUSHORT         Buffer,
  ULONG           Count
);
```

Parameters

`Register`

Pointer to the register, which must be a mapped range in memory space.

`Buffer`

Pointer to a buffer from which an array of USHORT values is to be written.

`Count`

Specifies the number of USHORT values to be written to the register.

Return Value

None

Remarks

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of **WRITE_REGISTER_BUFFER_USHORT** can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib

DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_UCHAR function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_UCHAR** routine writes a byte to the specified address.

Syntax

```
void WRITE_REGISTER_UCHAR(
  volatile UCHAR *Register,
  UCHAR          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_UCHAR** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_ULONG function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_ULONG** routine writes a ULONG value to the specified address.

Syntax

```
void WRITE_REGISTER_ULONG(
  volatile ULONG *Register,
  ULONG          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_ULONG** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)

WRITE_REGISTER_USHORT function

1/25/2019 • 2 minutes to read • Edit Online

The **WRITE_REGISTER_USHORT** routine writes a USHORT value to the specified address.

Syntax

```
void WRITE_REGISTER_USHORT(
  volatile USHORT *Register,
  USHORT          Value
);
```

Parameters

`Register`

`Value`

Return Value

None

Remarks

Callers of **WRITE_REGISTER_USHORT** can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

Requirements

Minimum supported client	Available starting with Windows 2000.
Target Platform	Universal
Header	wudfwdm.h (include Wdm.h, Ntddk.h, Ntifs.h, Ioaccess.h, Miniport.h, Wudfwdm.h)
Library	NtosKrnl.lib
DLL	NtosKrnl.exe
IRQL	Any level (see Remarks section)